TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JANI POHJALAINEN

VANHAN OHJELMISTON NOPEUTUS JA SIIRTO WEB-ARKKITEHTUURIIN

Diplomityö

Tarkastaja: Kari Systä
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunta-neuvoston kokouksessa 05. helmi-kuuta 2014

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
POHJALAINEN, JANI: Vanhan ohjelmiston nopeutus ja siirto web-arkkitehtuuriin
Diplomityö, 49 sivua, 6 liitesivua
Kesäkuu 2014
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Kari Systä
Avainsanat: Vanha ohjelmisto, ohjelmiston nopeutus, web-arkkitehtuurit, kuormantasaus, zmq, rinnakkaistaminen

Käyttäjien ja asiakkaiden kehittyvät vaatimukset ja uudet käyttömahdollisuudet tekevät monista yrityksille tärkeistä ohjelmista auttamattoman vanhoja. Näitä vanhoja ohjelmia ei voida hylätä, koska ne ovat keskeinen osa yrityksen it-infrastruktuuria. Vanhat ohjelmat voidaan joko suunnitella alusta asti täysin uudestaan tai ne voidaan kääriä uudemman ohjelman sisälle.

Tässä työssä käydään läpi yhden tällaisen vanhan ohjelmiston muunnos web-tekniikoille. Työssä käsitellään myös vanhan ohjelmiston nopeuttamista niin rinnakkaistamisella kuin web-tekniikoiden sallimilla kuormantasaus menetelmillä.

Työ on jaettu kolmeen osaan: teoriaosaan, työssä käsiteltävän ohjelmiston esittelyosaan ja toteutusosaan. Teoriaosassa käsitellään muutamia web-tekniikoita ja rinnakkaistamisen perusteita, lisäksi käydään läpi lyhyesti mitä ohjelmiston ylläpidossa pitää ottaa huomioon. Esittelyosassa työssä käsiteltävä ohjelmisto esitellään yleisellä tasolla ja samalla käydään lävitse niitä ongelmia, jotka ovat johtaneet ohjelmiston muuttamiseen. Toteutusosassa käydään lävitse toteutuksen kaksi pääasiallista haara, joita työn aikana käytiin läpi. Nämä haarat olivat etukäteen toteutettu laskenta ja reaaliaikainen laskenta. Lopuksi esitellään, minkälainen uudesta ohjelmistosta tehtiin ja kuinka paljon sitä on onnistuttu nopeuttamaan kuormantasauksella ja ohjelmointikielen vaihtamisella verrattuna vanhaan ohjelmistoon.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
POHJALAINEN, JANI: Improving the performance of legacy software and transferring it to web architecture
Master of Science Thesis, 49 pages, 6 Appendix pages
June 2014
Major: Software Engineering
Examiner: Professor Kari Systä
Keywords: Legacy software, software performance, web architectures, load balancing, zmq, parallelisation

Customers' and users' developing demands and new use case possibilities render many legacy software products obsolete. These legacy products cannot be just discarded because they are integral part of the companies' IT-infrastructure. One can either design these software products completely from the beginning or wrap them inside of a newer program.

This theses goes through the process of updating one of these kinds of legacy software to web architecture. Thesis also contains the process of improving the performance of the legacy software. This process has been done through parallelisation and load balancing.

This thesis is divided to three parts: theory part, presented legacy software part and work part. The theory part showcases few web techniques/architectures and also details the basics of software parallelisation, also little bit of program maintenance is discussed. The legacy software is detailed on its overall level and also some of the problems that has led to the changing of the legacy software are presented in the second part. In the work part of the thesis the two main paths that was taken during the conversion process are presented. These paths were pre-calculation and real-time calculation. In the end of the thesis the new software, and how much it has improved, is presented.

ALKUSANAT

Haluan kiittää Oy Ekocoilin Jouni Mäyrää diplomityön mahdollistamisesta, Jarkko Salmelinia avusta vanhan ohjelmiston ymmärtämisessä ja Tomi Pohjalaista kehitysympäristön rakentamisesta ja kehitystyön avustamisesta. Haluan myös kiittää työni tarkastajaa professori Kari Systää työn kommentoinnista ja neuvoista koko työnteon ajan.

Tampereella 12.5.2014

Jani Pohjalainen

SISÄLLYS

TERMIT, LYHENTEET JA NIIDEN MÄÄRITELMÄT

Termi	Selitys
Active Directory	Active directory (AD) on Microsoftin kehittämä käyttäjänhallintapalvelu Windows-toimialueelle.
API	Application Programming Interface. Rajapinta, jolla ohjelmat voivat keskustella keskenään.
Demilitarisoitu alue	Demilitarisoitu alue (Demilitarized zone DMZ) on verkkotekniikoissa käytetty termi, joka kuvaa aliverkkoa, joka yhdistää turvallisen sisäverkon turvattomaan ulkoverkkoon.
Dll	Lyhenne sanoista Dynamic Link Library. Dll on Microsoftin toteutus sovellusten välillä jaetusta koodista. Dll-koodia voi käyttää useampi ohjelma.
Jasper	Javalla tehty avoimen koodin raportointiohjelmisto.
JSON	Lyhenne sanoista JavaScript Object Notation. Avoimen standardin tiedostomuoto, jota käytetään tiedonvälitykseen.
Klusteri	Klusteri on useamman tietokoneen muodostama verkotettu malli. Klusterissa on yleensä yksi palvelin, joka jakaa muille klusterissa oleville koneille tehtäviä.
Kätkö	Kätkö on suomennos sanasta Cache. Muun muassa Internetissä käytetty tekniikka, joka vähentää tarvetta lukea tai hakea samaa tietoa aina uudestaan.
MVC	Ohjelmistoarkkitehtuuri. Lyhenne sanoista Model View Controller (Malli, näkymä ja käsittelijä).
ORM	Lyhenne sanoista Object-relational Mapping. Ohjelmistotekniikka, jossa luodaan muunnos kahden, ei yhteensopivan, tietorakenteen välillä. Esimerkiksi tietokannan ja ohjelmiston välille luotu luokka, joka yhdistää nämä kaksi toisiinsa.
Pragma	Ohjelmointikielissä olevia, vain kääntäjälle tarkoitettuja ohjeita joiden mukaan sen pitäisi koodia kääntää.
URI	Lyhenne sanoista Uniform Resource Identifier. Merkkijono, joka kuvaa jonkin tiedon paikan tai yksikäsitteisen nimen.
Zend Framework	PHP MVC-ohjelmistokehys.

1 JOHDANTO

Ohjelmistokehitys ja ohjelmistojen käyttäjien tarpeet ovat kehittyneet hyvin nopeasti viime vuosien aikana. Tästä kehitys nopeudesta johtuen jo kymmenen vuotta sitten tehdyt ohjelmistot voivat olla auttamattomasti vanhentuneita, puhumattakaan yli kaksikymmentä vuotta sitten tehdyistä ohjelmistoista.

Vanhat ohjelmistot eivät pysty enää vastaamaan käyttäjien uusiin vaatimuksiin, eikä yritysten uusiin toimintamalleihin ja businessvaatimuksiin. Tällaisten ohjelmistojen kohdalla täytyy miettiä ohjelmiston päivittämistä uuteen aikaan. Varsinkin Internet on luonut paljon uusia käyttömahdollisuuksia ja –haasteita, siksipä on hyvin houkuttelevaa muuntaa vanha ohjelma web-tekniikoita käyttäväksi ohjelmaksi.

Web-ohjelmat antavat ohjelmien käyttäjille suuria vapauksia käyttämisen ajan ja paikan suhteen, mutta nämä vapaudet aiheuttavat ongelmia ohjelmien tekijöille ja ylläpitäjille. Web-ohjelmat myös asettavat erilaisia vaatimuksia ohjelmistojen ajoympäristöille kuin normaalit työpöytäsovellukset.

Web-sovelluksia tehtäessä täytyy ottaa huomioon niin asiakaspään ohjelmointi, yleensä verkkoselain, ja palvelinpään ohjelmointi. Lisäksi web-tekniikat kehittyvät edelleen hyvin nopeasti, ja verkosta löytyy sadoittain erilaisia ohjelmistokehyksiä ja useita ohjelmointikieliä, joilla web-sovelluksia voi tehdä. Näistä ohjelmistokehyksistä omaan projektiin oikean löytäminen ei ole helppoa.

Toinen suuri muutos, joka on tapahtunut viimeisen kahdenkymmenen vuoden aikana, on prosessoreiden ytimien lukumäärän kasvu. Tämä kasvu mahdollistaa kaikille ohjelmistonkehittäjille ohjelman aidon rinnakkaistamismahdollisuuden. Rinnakkaistaminen auttaa ohjelmia toimimaan huomattavasti nopeammin, mutta myös vaikeuttaa ohjelmien suunnittelua ja toteutusta.

Vanha ohjelma voidaan muuntaa uudeksi usealla eri tekniikalla. Se voidaan kääriä toisen ohjelman sisälle, ja tällöin vanhaan ohjelmaan ei tarvitse juurikaan kajota. Kääriminen nopeuttaa uuden ohjelman toimintaan saamista, mutta tuo vanhan ohjelman ongelmat mukanaan uuteen ohjelmaan. Vanha ohjelma voidaan myös tehdä alusta asti uudelleen, jolloin siihen voidaan käyttää moderneja ohjelmointitekniikoita ja moderneja ohjelmointikieliä ja -kehyksiä. Tämä uudelleen kirjoittaminen tietenkin hidastaa uuden

ohjelman toimintaan saantia. Ohjelman uudelleen kirjoittaminen avaa myös mahdollisuuden ohjelman ylläpidettävyyden parantamisen.

Tässä työssä käydään läpi yhden vanhan ohjelman muuntamisprosessi web-tekniikoille. Luvussa kaksi esitetään tavoitteet, joita uudelle ohjelmalle asetettiin. Luvussa kolme esitellään web-tekniikoita ja arkkitehtuureita, siinä kuvataan myös eroja perinteisen työpöytäsovelluksen ja web-sovelluksen välillä. Kolmannessa luvussa käsitellään myös lyhyesti ohjelmiston ylläpitoa ja rinnakkaistamista. Neljännessä luvussa esitellään vanha ohjelma, jota tässä työssä esiteltävässä projektissa ollaan muuttamassa. Luvussa viisi on esitetty yksi suurimmista ongelmista vanhassa ohjelmassa ja miten sitä lähdettiin ratkomaan. Kuudesluku kuvaa uuden ohjelman tämän hetkisen tilanteen ja miten projektia tullaan jatkamaan. Seitsemänteen lukuun on kerätty johtopäätelmiä, joita projektin aikana on tehty. Kahdeksas luku on yhteenveto.

2 TAVOITTEET

Tämä luku esittelee kehitettävälle ohjelmalle asetetut tavoitteet. Nämä tavoitteet toimivat kehyksenä ja vaatimuslistana uuden ohjelman suunnittelulle ja toteutukselle.

2.1 Uuden ohjelman tavoitteet

Uudelle ohjelmalle asetettiin muutamia tavoitteita, jotka sen piti täyttää. Näitä oli muun muassa: helppokäyttöisyys, nopeus, ylläpidettävyys ja saavutettavuus.

Helppokäyttöisyystavoite asetettiin johtuen siitä, että ohjelmaa on tarkoitus tarjota asiakkaille, eikä vain oman organisaation jäsenille. Asiakkaat, vaikka usein ovatkin jossain määrin alaan perehtyneitä, eivät ole asiantuntijoita. Asiakkaan, tiedoista ja taidoista riippumatta, olisi kuitenkin pystyttävä tekemään tarvitsemansa mitoituslaskennat ohjelmalla.

Oman organisaation ulkopuolisten käyttäjien takia ohjelmistoon täytyi myös saada käyttäjänhallinta. Tällä tarkoitetaan niin käyttäjän tunnistamista kuin autorisointia, eli jokaisen käyttäjän käyttöoikeuksien valvontaa.

Vanhasta ohjelmasta on saatu sitä käyttäviltä ihmisiltä palautetta, että sen suorittamat laskennat kestävät liian kauan. Suurimmat muutokset itse laskentaosaan ohjelmasta keskittyvätkin tämän nopeutustavoitteen saavuttamiseen. Koettiin, että nopeustavoite oli yksi tärkeimmistä saavuttaa, joten sen toteuttamiseen käytettiin paljon aikaa. Ohjelman nopeuttamista käsitellään tarkemmin luvussa viisi.

Viimeisenä tärkeänä tavoitteena on ohjelman saavutettavuus. Tällä tarkoitetaan sitä, että ohjelma olisi kaikkien sitä tarvitsevien käytettävissä, missä ja milloin tahansa. Tavoitteen saavuttamiseksi uudeksi ohjelman toteutustekniikaksi valittiin verkkopohjaiset tekniikat, jolloin ohjelmaa ei tarvitsisi erikseen asentaa sitä käyttävän henkilön koneelle, vaan riittää, että käyttäjälle on luotu tunnukset yrityksen kotisivuille. Kotisivuilta hän voi kirjautua ohjelmaan ja käyttää sitä kaikkialla missä on Internet-yhteys. Internet-yhteyden vaatiminen kuitenkin rajoittaa saavutettavuutta, mutta tätä rajoitusta ei enää nykyaikana pidetty niin rajoittavana, jotta ohjelmasta olisi tehty myös työpöytäversio.

Yksi tavoite oli myös parantaa ohjelmiston ylläpidettävyyttä ja päivitettävyyttä. Nämä tavoitteet eivät kuitenkaan olleet alusta alkaen korkealla tavoitelistalla, joten näitä tavoitteita pyrittiin ratkaisemaan ohjelmiston kehityksen loppuvaiheilla.

2.2 Projektin tavoitteet

Projektin tavoitteena ei ole vain saada aikaiseksi uusi ohjelma, joka vastaa edellä asetettuihin ohjelmallisiin vaatimuksiin, vaan sellainen ohjelmisto, joka on myös tulevaisuudessa helppo muuttaa vastaamaan muuttuneisiin tarpeisiin. Valmiin ohjelmiston tulee olla myös testattu hyvin ja kattavasti, ei vain ohjelman käyttäytymisen osalta, mutta myös ohjelman businesslogiikan osalta. Jatkossa on voitava pystyä luottamaan siihen, että ohjelmisto laskee ja mitoittaa tuotteet oikein, ja kun uusia tuotteita lisätään, myös niiden laskentaan pitää pystyä luottamaan ilman suuria lisäpanostuksia testaamiseen.

3 TAUSTATUTKIMUS

Tämän luvun tarkoituksena on käsitellä minkälaisille tekniikoille ja malleille uusi ohjelma tulee pohjautumaan. Luvussa käsitellään myös lyhyesti mitä eroavaisuuksia löydettiin perinteisen työpöytäsovelluksen tekemisessä ja suunnittelussa, verkkopohjaisen sovellukseen verrattuna. Näiden lisäksi luvussa käydään läpi perusteoriaa niin ohjelmistojen rinnakkaistamisesta kuin ylläpidettävyydestä.

3.1 Internetissä käytettyjä tekniikoita ja arkkitehtuureita

Alun perin Internet oli suunniteltu yksinkertaiseksi staattisten verkkosivujen hakupalveluksi. Tästä johtuen Internetin perusrakenne on niin kutsuttu asiakas-palvelin -malli. Asiakas-palvelin -mallissa asiakas pyytää palvelimelta haluttuja resursseja, kuten verkkosivuja, ja palvelin vastaa asiakkaan pyyntöön palauttamalla pyydetyn resurssin. Tämä malli johtaa yhteyden tilattomuuteen, eli palvelin ei tiedä asiakkaasta mitään muuta, kuin yhden pyynnön.

Internetin perustasta aiheutuva tilattomuus tuo käyttäjien erottamisongelman muista käyttäjistä. Tämä ongelma näkyy helpoiten tunnistautumista vaativissa verkkosivustoissa. Kuinka voidaan tietää onko käyttäjän sallittu käyttää kyseistä sivustoa tai osaa sivustosta? Yleisesti tämä ongelma ratkaistaan kekseillä.

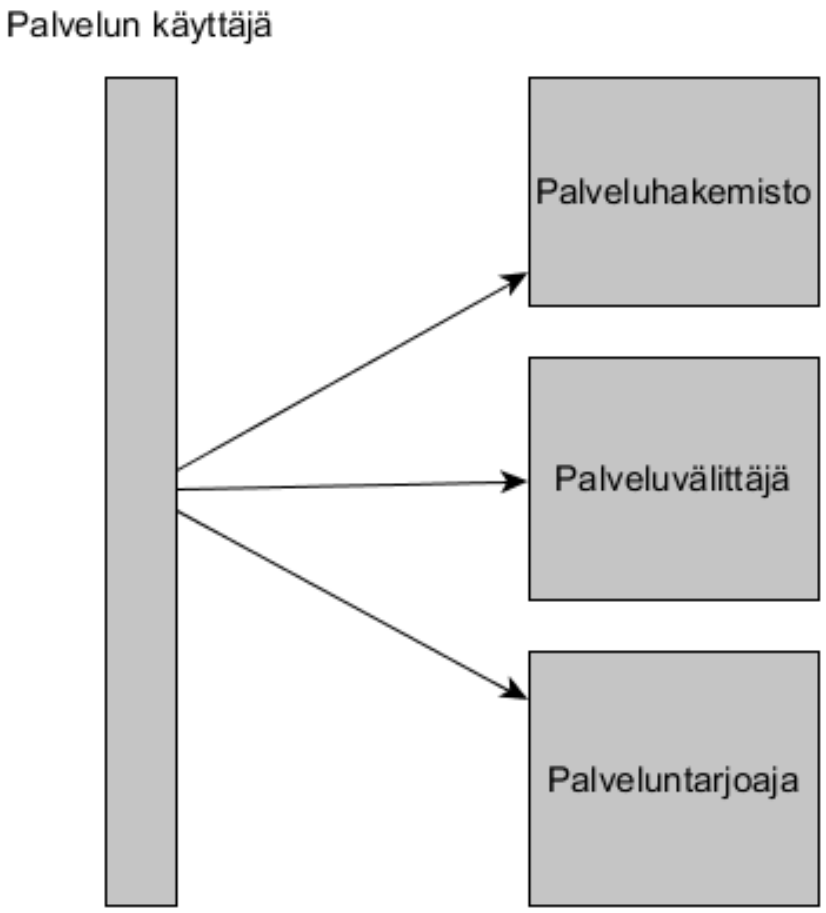
Istunnossa jokaiselle käyttäjälle luodaan oma tunnistetieto, jota kuljetetaan niin kutsutuissa kekseissä, jokaisen uuden pyynnön mukana. Palvelin voi sitten tarkistaa tästä keksissä olevasta tunnisteesta onko kyseessä oikea käyttäjä ja onko hänellä oikeus tehdä kyseistä pyyntöä. Istuntoihin ja kekseihin liittyy monia tietoturvaongelmia, mutta ne eivät ole tämän työn aihepiirissä. Palvelin voi myös ylläpitää käyttäjästä joitakin haluttuja tietoja. Nämä tiedot ovat myös tallennettuna käyttäjän istuntotietoihin. Tämä tieto pidetään yleisesti palvelimella.

Monessa tapauksessa tällainen pyyntö–vastaus malli on täysin toimiva ja riittävä ratkaisu, mutta jos halutaan jäljitellä työpöytäsovelluksen interaktiivisuutta tämä aiheuttaa ongelmia. Työpöytäohjelman ja web-ohjelman eroista on tarkemmin aliluvussa 3.1.3.

3.1.1 Web-service, REST ja SOAP

Yksi palvelin-asiakas perusarkkitehtuurista johdettu arkkitehtuuri on palvelukeskeinen arkkitehtuuri (Service oriented architecture SOA). SOA on yksinkertaisesti esitettynä arkkitehtuuri, jossa jokin palvelu tarjotaan kenelle tahansa, jolla on oikeus käyttää kyseistä palvelua Internetin ylitse. Palvelu tarjotaan käyttäjille rajapintana, jolle palvelun käyttäjä ohjelmoi rajapintaa käyttävän osan. [16.] Yleisesti palvelun rajapinta kuvataan Web Service Description Language:lla (WSDL). Tämä on W3C:n määrittämä XML-pohjainen kieli. [19.]

Hieman tarkemmin määriteltynä SOA:ssa palvelut ovat löydettäviä palveluita, jotka toimivat viestinvälitys-mallin mukaisesti [3]. Kuvassa 3.1 on esitettynä korkean tason arkkitehtuurikuva SOA-palvelusta.



Kuva 3.1: SOA termit ja korkean tason arkkitehtuuri. [Mukailtu [3], kuva 2-3, s. 22]

Palvelun käyttäjät ovat ohjelmia, jotka kutsuvat palveluntarjoajaa, eli niitä ohjelmia tai käyttäjiä, jotka ovat sallittu käyttää tarjottavia palveluita. Palveluhakemisto on rekisteri johon tarjottavat palvelut rekisteröityvät, jotta ne ovat kaikkien löydettävissä. Palveluvälittäjä on taas erikoistunut ohjelma, joka välittää palvelupyyntöjä palveluntarjoajille. Palveluntarjoaja on ohjelma, joka toteuttaa palvelun. [3.]

SOA:n käyttämisessä on kaksi suurta ongelmaa, jotka ovat aika ja monimutkaisuus [16]. Suunniteltaessa tarjottavaa palvelua täytyy kehittäjien ottaa huomioon, että verkon yli lähetetyt pyynnöt palveluun tulevat kärsimään verkkoviiveestä. Tämä viive voi olla muutamasta sadasta millisekunnista jopa sekunteihin riippuen käyttäjän verkkoyhteydestä. Verkkoviiveeseen vaikuttaa myös se kuinka paljon parametreja pyynnön yhteydessä lähetetään ja kuinka paljon vastauksia pyyntöön vastattaessa lähetetään.

Monimutkaisuus näkyy SOA:ssa rajapinnan kokona. Jos parametreja ja vastauksia on paljon, on rajapinnan toteuttaminen lähempänä ohjelmaa, kuin vain ohjelman osaa. Tämä johtaa siihen, että käyttäjän kannattaa alkaa harkita kannattaisiko hänen itse ohjelmoida palvelun toteutus. Suurien rajapintojen testaaminen on myös vaikeaa ja vaatii erikoistunutta testaustietoutta. [16.]

Näistä kahdesta ongelmasta johtuen, kun ollaan suunnittelemassa SOA-pohjaista järjestelmää, pitää pyrkiä mahdollisimman pieneen parametrien määrään ja mahdollisimman pieneen vastausten joukkoon. Mieluummin pitäisi tarjota useita pieniä palveluita kuin muutamaa isoa palvelua. [16.]

SOA on enemmänkin palvelinpäänarkkitehtuurimalli eikä se ota kantaa asiakas/käyttäjäpään osaan. Tässä työssä käsiteltävän ohjelman tapauksessa SOA onkin otettu käyttöön ideatasolla laskennan toteuttamiseen, mutta sitä on hyvin paljon yksinkertaistettu jättämällä pois palvelunetsimet ja jossain määrin myös palvelunvälittimet. Näitä osia ei tarvittu, koska palvelua ei haluta tarjota, kuin niille käyttäjille, jotka ovat yrityksen hyväksymiä. Palvelun käyttäjien joukko tulee olemaan niin rajattu, että niiden tekijöille voidaan ilmoittaa palvelun osoite suoraan ilman, että heidän täytyisi mennä palvelunvälittimen kautta.

Yksi yleisesti käytössä oleva malli, joka ottaa kantaa asiakkaan ja serverin väliseen yhteyteen, on Representational State Transfer eli REST. Tämä malli pohjautuu olemassa oleviin rajoitteisiin, joita on puhdas asiakas-palvelin –malli, jossa näyttölogiikka erotetaan kokonaan datan hallinasta. Tämä erotus parantaa sovelluksen näyttöjen siirrettävyyttä muille alustoille. Tilattomuus on toinen suuri rajoite REST-mallissa. Tilattomuus rajoite tarkoittaa, että jokaisen pyynnön palvelimelle pitää sisältää kaikki tarvittava tieto, jotta pyyntöön voidaan vastata. Palvelimen ei tarvitse tietää mitään edellisistä pyynnöistä tai käyttäjän toimista. Tämä johtaa siihen, että käyttäjän tila pitää pitää tiedossa

puhtaasti asiakkaalla. Näiden kahden rajoitteen lisäksi REST pohjautuu vielä muihinkin rajoitteisiin kuten kätköihin ja yhtenäisiin rajapintoihin. [4.]

Yleisesti nähtävä REST-toteutus on käyttäjälle tarjottu rajapinta, joka toteuttaa yhteen URI:in tehtävien erilaisten http-pyyntöjen kautta useita toiminnallisuuksia. Esimerkiksi: tarjotaan osoitetta http://esimerkki.fi/esimerkki, joka toteuttaa http GET-pyynnölle eri toiminnallisuuden, kuin http POST-pyynnölle. Yleensä get:n ja post:n lisäksi tarjotaan put ja delete toiminnallisuudet. Tällaista rajapintaa kutsutaan *RESTful*-rajapinnaksi.

Ongelmaksi RESTful-api:ssa tuli käytetyn ohjelmistokehyksen tuen puute RESTful-rajapinnan tekoon. Uudemmat versiot käytetystä ohjelmistokehyksestä tukevat kyllä RESTful-rajapinnan tekoa.

Toinen yleinen malli, kahden palvelun väliselle tiedonvälitykselle verkon yli, on Simple Object Access Protocol (SOAP). SOAP on keino määritellä sääntöjä XML-pohjaiselle viestinnälle. SOAP voi käyttää siirtotienään http:tä tai SMTP:tä. SOAP tukee useita eri protokollia ja teknologioita yhteyden määrittelyyn, muun muassa WSDL:ää ja XSDs:ää [2]. SOAP ei ole tilaton, kuten REST. REST-rajapinta on kuitenkin yksinkertaisempi toteuttaa, kuin SOAP-rajapinta, joka vaatii rajapinnan kuvauksen WSDL:lä tai vastaavalla. Johtuen tästä REST:in yksinkertaisuudesta ja määrittelemättömyydestä, täytyy asiakkaalla ja palveluntarjoajalla olla hyvä yhteisymmärrys palvelun käytöstä. Yksi SOAP:in vahvuus on sen sisältämä virheenkäsittely, tätä REST ei oletuksena tarjoa. [12.]

3.1.2 Websocket ja Ajax

Websocket:it on kehitetty tarjoamaan netissä toimiville sovelluksille tehokkaan tilallisen yhteyden. Niiden ei tarvitse lähettää jokaisen pyynnön ja vastauksen yhteydessä http-otsikoita uudestaan vaan ne toimivat yhdessä portissa, vähentäen turhan datan lähettämistä. [17.] Websocket tarjoaa siis kaksisuuntaisen yhteyden asiakas sovelluksen ja palvelimen välillä [14].

Asynchronous Java Script and XML (AJAX) on kokoelma tekniikoita, joilla pystytään luomaan asynkronisia web-ohjelmia. Pyyntöjen tekemiseen käytetään XMLHttpRequest-objekteja, jotka ovat ECMAScriptin http-api:n määrittelemiä [20]. Jokaisen pyynnön ja vastauksen mukana kulkevat http-otsikot, eli ne vastaavat monilta osin normaalia web-kommunikointia.

Syy miksi websocket:ja ei otettu käyttöön uuden ohjelman teossa oli yksinkertaisesti niiden varhainen kehitysvaihe. Ne ovat tällä hetkellä vielä luonnosvaiheessa, niistä ei

siis vielä ole standardia. Suurin osa nykyaikaisista selaimista kuitenkin toteuttavat niistä RFC version 6455 [18].

Ajax on jo useita vuosia käytössä ollut tekniikka ja vaikka se kuluttaakin kaistaa, niin sanotusti turhaan, ei se työssä esiteltävän ohjelman tapauksessa ole suuri ongelma. Ajax ohjelmointi on myös helpompaa tehdä perinteisillä JavaScript kirjastoilla kuin websocket-ohjelmointi.

3.1.3 Työpöytäsovellus vs. websovellus

Työpöytäsovelluksen ja websovelluksen suunnittelussa ja testauksessa on otettava huomioon muutama Internetissä toimivan ohjelmiston erikoisuus, kuten asiakas- ja palvelinohjelmoinnin eroavuus. Internet-sovelluksessa täytyy erikseen ohjelmoida asiakassovellus ja palvelinsovellus, usein vielä eri ohjelmointikielillä. Työpöytäsovelluksessa ne ovat usein yksi ja sama ohjelma. Vaikka kummassakin ohjelmointiympäristössä voidaan käyttää samanlaisia ohjelmistomalleja, kuten Model View Controller-mallia (MVC), on niissä joitakin eroavaisuuksia.

MVC-mallissa tarkoituksena on erottaa näyttölogiikka, *view*, datasta, *model*, koska näyttölogiikka on alttiimpi muuttumiselle kuin datanhallintalogiikka. MVC:ssä on vielä kolmas komponentti *kontrolleri* tai *käsittelijä*, jonka tarkoituksena on yhdistää näytöt dataan. Nämä *kontrollerit* sisältävät käyttäjäinteraktioiden hallinnan lisäksi myös businesslogiikkaa, joka ei suoraan ole tekemisissä datan käsittelyn kanssa. [8.]

Yleisesti verkko-ohjelmissa näytöt eivät suoraan voi olla yhteydessä *malliin*, vaan niiden täytyy mennä *kontrollereiden* kautta. Verkko-ohjelmat toteuttavatkin usein niin sanotun passiivisen MVC-mallin [11]. Työpöytä puolella näytöt useasti voivat suoraan tallentaa dataa *malliin* tai saavat suoraan ilmoituksen, kun data on muuttunut ja näyttö pitää päivittää. Tämä on niin sanottu aktiivinen MVC-malli [11]. Verkko-ohjelmissa aktiivista MVC-mallia voidaan nykyään toteuttaa käyttämällä esimerkiksi websocket:ja.

Työssä esiteltävä ohjelmisto käyttää ohjelmistokehyksenään Zend Framework versiota 1. Tämä kehys on suunniteltu nimenomaan passiivisen MVC-mallin mukaiseksi. [21.]

Ohjelman verkkopohjaisuus helpottaa myös koko ohjelman päivittämistä. Normaalissa työpöytäsovelluksessa ohjelman täytyy suorittaa, joko käyttäjän toimesta tai itsenäisesti, tarkastus onko ohjelmasta tarjolla uudempaa versiota. Mutta verkkosovellutus voidaan päivittää palvelimille valmistajan toimesta ilman, että asiakkaan tarvitsee tehdä mitään tai, että asiakas edes huomaa päivitystä. Tämä päivittämisen nopeus ja helppous oli yksi isoista verkkosovelluksen hyvistä puolista.

Ohjelmien testaus on erittäin tärkeää ja siihen on olemassa paljon automatisointityökaluja. Verkkopohjaisen ohjelman testaaminen on kuitenkin vaikeampaa kuin normaalin

työpöytäohjelmiston. Tämä vaikeus johtuu useista eri ohjelmointikielistä ja siitä, että verkkosovellusta ajetaan palvelimen päällä eikä itsenäisesti kehityskoneelta. Vaikeutta lisää myös monivaiheisempi testaus.

Testausvaiheita tässä projektissa oli PHP:n testaaminen PHPUnit:n avulla, JavaScript:n testaaminen Jasminen avulla ja näyttöjen ja kokonaistoiminnallisuuden testaaminen Webdriver:llä/Selenium 2.0 avulla. Testaus tapahtui etäpalvelimella, jossa ohjelman kehitysversiota ajettiin Zend-serverin päällä.

Yksikkötestaaminen on melkein samanlaista kuin normaalin työpöytäohjelman yksikkötestaaminenkin. Eroavaisuuksia on useamman ohjelmointikielen testaaminen ja JavaScript:n tapauksessa yksikkötestauksen sotkeutuminen myös näyttöjen testaamiseen. Työssä käytetty Jasmine ei tarvitse verkkosivua testiä ajaakseen, kuten jotkut muut JavaScript testityökalut vaativat. Näin ollen testejä varten ei tarvinnut tehdä uutta sivua tai laittaa testauskoodia tuotantosivustoihin.

Jasmine on niin kutsuttu käyttäytymiseen perustuvaa testaamista. Siinä kirjoitetaan testit helposti luettavaan muotoon, jotta testien tarkoitus kävisi selväksi lukijalle heti ilman koodiin perehtymistä. Esimerkki Jasminen testistä on esitettynä listauksessa 3.1.

```
describe("Testi kokoelma", function() {
it("Tässä olisi testiä kuvaava lause", function() {
expect(true).toBe(true);
  });
});
```

Ohjelma 3.1: Esimerkki Jasmine testistä

Näyttöjen toiminnallisuuden testaamiseen lisävaikeutta toi laaja AJAX:n käyttö, koska monet näytön elementit ladataan vastauksena käyttäjän toimiin. Tästä elementtien asynkronisesta lataamisesta johtuen täytyi testien tietää kuinka kauan oli sallittua odottaa uutta elementtiä. Perinteisellä tavalla tehtynä testeistä olisi tullut jatkuvien ajastimien sekamelskaa ja niiden ajaminen olisi kestänyt kauemmin kuin olisi haluttu. Selenium:ssa tämä AJAX:n odottamisongelman voi kiertää ajamalla JavaScript koodia suoraan testistä. Esimerkki AJAX ongelman kiertämisestä on esitettynä ohjelmassa 3.2.

```
/**
  * Waits for all AJAX calls to end or timeout's in given timeout.
  * @param timeout given in seconds. Default 5 seconds
  * @param driver
  */
 public void WaitForAjax(inttimeout,WebDriverdriver){
   // Get a JavascriptExecutor
   JavascriptExecutorexec=(JavascriptExecutor)driver;

   for(inti=0;i<timeout;i++){
     try{
       Thread.sleep(1000);
       if((Boolean)exec.executeScript("return jQuery.active == 0"))
         break;
     }catch(InterruptedExceptione){
       e.printStackTrace();
       thrownewIllegalStateException("Ajax timed out after waiting for
"+timeout+" seconds");
     }
   }
 }
```

Ohjelma 3.2: Esimerkki toteutus AJAX tarkastuksesta Javalla toteutettuna

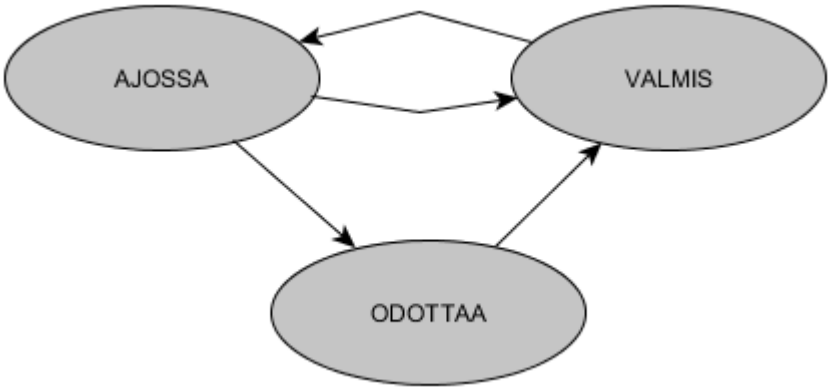
Tämä funktio tarkistaa suoraan jQuery JavaScript-kirjastolta onko sillä enää yhtään AJAX kutsua suorituksessa, ja jos sillä on odottamassa AJAX kutsuja, jää testi odottamaan niiden valmistumista tai testi hylkää itsensä aikakatkaisun tullessa täyteen.

Integraatiotestaus on myös erilaista normaaliin työpöytäsovelluksen testaamiseen verrattuna, koska se täytyy suorittaa kaikissa tuetuissa selaimissa ja kaikissa tuetuissa käyttöjärjestelmissä. Integraatiotestaamiseen on myös olemassa avustavia työkaluja kuten Selenium:in RC, joka on käytännössä palvelin, joka ajaa testit useammassa prosessissa eri selaimilla yhtä aikaa [15].Selenium 2 tarjoaa myös Selenium Grid nimisen toiminnallisuuden, jonka avulla testien ajamista voidaan rinnakkaistaa ja testejä voidaan ajaa useammalla selaimella ja käyttöjärjestelmällä. Selenium 2 Grid muistuttaa toiminnaltaan paljon jQuerynTestSwarm:a.

Toinen työkalu on jQueryn käyttämä TestSwarm. TestSwarm:i toimii niin, että käyttäjät liittyvät TestSwarm:in palvelimeen ja antavat oman selaimensa TestSwarm:in käyttöön [9]. Ongelmana tässä lähestymistavassa yksityiselle yritykselle on se, että yrityksen sisälle pitäisi rakentaa useita koneita, joissa on eri selaimia, joita TestSwarm voisi käyttää. Tämä testaustapa myös vaatii enemmän käsityötä kuin esimerkiksi Selenium RC, koska jokainen asiakaskone pitää erikseen käydä liittämässä TestSwarm:iin ja jokaista asiakaskonetta täytyy myös ylläpitää.

3.2 Rinnakkaistaminen

Prosessi on ohjelma, joka on ajossa tietokoneella. Prosessi sisältää muun muassa tiedon siitä onko kyseinen prosessi missä tilassa. Näitä prosessin tiloja on *ajossa, odottaa* ja *valmis*. Yhden suorittimen järjestelmässä ei tietenkään ajossa voi olla kuin yksi prosessi kerrallaan ja moniydin järjestelmässä niin monta kuin ytimiä prosessorissa on. [5.] Perustason prosessikuva on esitettynä kuvassa 3.2. Kuvassa esitetään prosessin kolme tilaa ja niiden väliset siirtymät.



Kuva 3.2: Prosessin tilakaavio [5 s.50 kuva 2.1]

Nykyaikaisissa prosessoreissa, varsinkin Intelin valmistamissa, puhutaan usein säikeistä pelkkien ytimien sijaan. Säikeen ja ytimen ero on siinä, että säie on kevyempi versio prosessista, näin ollen ne vaihtuvat nopeammin kuin prosessit ja vähempi ytiminen prosessori voi ajaa niitä siis useampia. [5.]

Rinnakkainen ohjelma käyttää useita, joko säikeitä tai prosesseja, jotka kommunikoivat keskenään tai käyttävät yhteisiä tietorakenteita. Todellisessa rinnakkaisessa ohjelmassa nämä säikeet tai prosessit suoritetaan yhtä aikaa.

3.2.1 Rinnakkaisuuden tuomia ongelmia

Rinnakkaisuus tuo ohjelmiin neljä pääongelmaa: *poissulkeminen, synkronointi, lukkiutuminen* ja *nälkiintyminen*.

Poissulkemisella tarkoitetaan ongelmaa joka muodostuu prosessien käyttämien yhteisten tietorakenteiden päivittämisestä yhtä aikaa. *Semafori* on yksi keino estää poissulkemisongelma. Semaforeille on monia eri ratkaisuja, mutta yksinkertaisuudessaan niillä pyritään lukitsemaan jokin tietty kriittinen alue, esimerkiksi tiedon tallennus jaettuun muuttujaan. [5.]

Synkronoinnilla tarkoitetaan ongelmaa joka ilmaantuu kun prosessit pitäisi tahdistaa toistensa kanssa [5]. Tämä ongelma voidaan yrittää ratkaista käyttämällä ajastettuja odotuksia joissa prosessi odottaa annetun ajan tiedon saantia, jonka jälkeen se joko tekee virheen tai jatkaa toimintaansa.

Lukkiutumisella tarkoitetaan ongelmaa jossa esimerkiksi prosessit odottavat toisen prosessin tulosta, ja tämä toinen prosessi odottaa taas toisen prosessin tulosta ja niin edelleen. Lukkiutuminen voidaan joko laukaista, välttää tai estää. Laukaiseminen on ohjelmallisesti hankalaa ja se onkin yleensä käyttäjän vastuulla tuhota jokin lukossa oleva prosessi. Välttämisessä prosessin tarvitsemat resurssit voidaan varata kaikki heti aluksi tai erilaisilla algoritmeilla prosessin aikana. Lukkiutumisen estäminen tehdään poistamalla jokin lukkiutumisen välttämättömistä ehdoista. [5.]

Nälkiintymisellä tarkoitetaan tilannetta jossa prosessi voisi jatkaa, mutta se ei jostain syystä saa suoritusaikaa. Nälkiintyminen voidaan estää lisäämällä prosesseille synkronointi kohtia, priorisoinnin vaihtelulla ja käyttämällä jonoissa FIFO-menetelmää. [5.]

3.2.2 Ohjelmien rinnakkaistaminen

Ohjelmia voidaan rinnakkaistaa useilla eri tavoilla, esimerkiksi käyttämällä erilaisia valmiita rinnakkaistamiskirjastoja, kuten Intelin Threading Building Blocksia tai C++:n Boost kirjastoa. Nämä kirjastot antavat ohjelmoijille valmiita funktioita ja olioita hoitamaan rinnakkaistamisen ongelmia. Huolimatta valmiista funktioista, ohjelmoija joutuu itse miettimään ja ratkaisemaan esimerkiksi kriittisten alueiden lukitsemisen.

Toinen vaihtoehto on käyttää korkeamman tason kirjastoja, kuten OpenMP:tä. OpenMP:ssä ohjelmoija määrittelee, esimerkiksi jonkin silmukkarakenteen, joka voidaan rinnakkaistaa, käyttämällä C-kielistä löytyviä pragmoja. Näillä pragmoilla käyttäjä ilmaisee kääntäjälle, että hän haluaa kääntäjän muuntavan rajatun alueen rinnakkaiseksi. Kääntäjä tekee tämän muunnoksen ilman muita ohjelmoijan toimenpiteitä.

Tässä työssä on rinnakkaistaminen toteutettu hieman erilailla kuin perinteisesti rinnakkaistamisella ymmärretään. Tämä rinnakkaistamistapa on kuvattu tarkemmin aliluvussa 5.3. Tähän tapaan päädyttiin, koska ohjelmalla suoritettiin nopeustestejä perinteisellä rinnakkaistamisella, jotka ovat kuvattuna aliluvussa 5.4

3.3 Ylläpidettävyys ja päivitettävyys

Ohjelmiston ylläpito on perinteisesti mielletty yhdeksi osaksi ohjelmiston elinkaarta. Ylläpito voidaan jakaa kolmeen tai neljään pienenpään osaan, osien ohjelmistoon kohdistamien syiden mukaan. Nämä osa alueet ovat Harsun mukaan *korjaava ylläpito, mukauttava ylläpito, täydellistävä ylläpito* ja *ehkäisevä ylläpito*. [6.]

Korjaava ylläpito:
Korjaava ylläpito sisältää virheiden, joita ei vielä testausvaiheessa ole huomattu, korjaamista. Tätä ylläpito-osaa voidaan helpottaa parantamalla ohjelmiston testattavuutta, kirjoittamalla modulaarisempaa koodia ja vähemmän toisistaan riippuvia funktioita, joita on sitten helpompi yksikkötestata.

Mukauttava ylläpito:
Tietokoneiden ja muiden teknisten laitteiden uusiminen on suhteellisen nopeaa ja ohjelmiston pitäisi kuitenkin toimia hyvässä tapauksessa useita vuosikymmeniä [6]. Muutokset ovat vieläkin nopeampia Internetissä ja sen käyttämissä tekniikoissa. Tähän muuttumiseen on kuitenkin melkoisen vaikeata ohjelmiston tasolla varautua. Yksi keino on pyrkiä poistamaan koodista mahdollisimman paljon ulkoisia riippuvuuksia, esimerkiksi käyttöjärjestelmän kirjastoihin. Toinen keino suojautua muutoksia vastaan on käyttää vakiintuneita ja paljon käytettyjä ohjelmointikieliä.

Täydellistävä ylläpito:
Uusien ominaisuuksien ja tietojen lisääminen muuttuvien käyttäjätarpeiden ja markkinatilanteiden mukaan on täydellistävää ylläpitoa. Esimerkkinä tällaisista muuttuvista tiedoista on tuotteiden hintatiedot ja uudet alituotteet, kuten esiteltävän ohjelmiston tapauksessa puhaltimet.

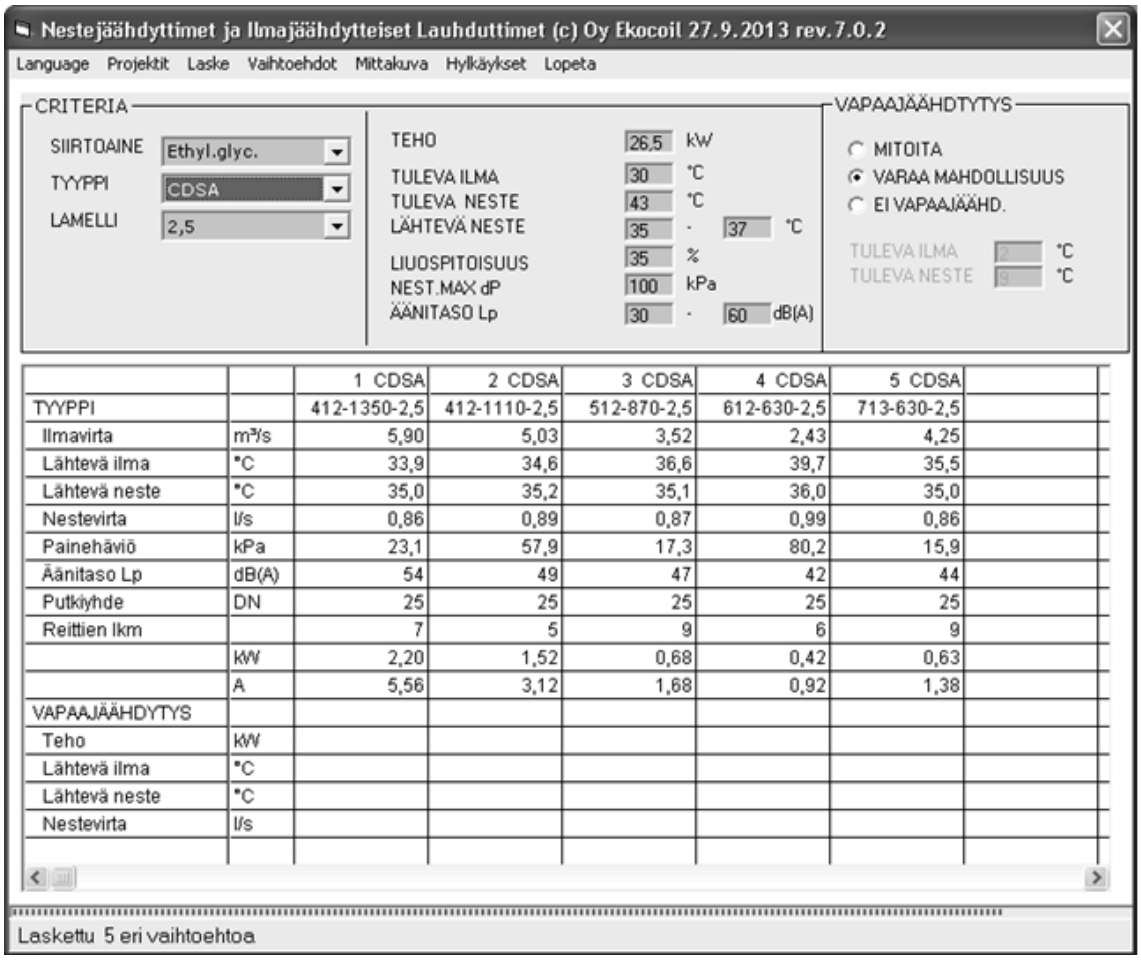
Ehkäisevä ylläpito:
Ehkäisevä ylläpito on ohjelmiston rakenteiden muuttamista ja korjaamista, jotta tulevat muut ylläpitotoimet olisivat helpompia ja nopeampia tehdä [6]. Ehkäisevä ylläpito ja täydellistävä ylläpito liittyvät helposti toisiinsa. Jos tulevaisuudessa tehtävää täydellistävää ylläpitoa halutaan ehkäistä, tai ainakin tehdä mahdollisimman helpoksi, kannattaa niiden tekeminen tehdä mahdollisimman helpoksi jo ohjelmiston kehitysvaiheessa.

4 OLEMASSAOLEVA OHJELMA

Tämä kappale kuvaa tällä hetkellä asiakkaiden ja myyjien käytössä olevan ohjelman rakenteen. Tämän lisäksi luvussa käsitellään minkälaisia osia ohjelmistosta on muunnettu, ja minkälaisia kehityspolkuja käytiin läpi muunnoksen aikana.

4.1 Ohjelma ja sen rakenne

Ohjelman käyttötarkoitus on laskea asiakkaan antamien parametrien mukainen jäähdytys- tai lauhdutinkone. Käyttäjä (yleensä myyjä) syöttää parametrit taulukkolaskentaohjelmaa muistuttavaan käyttöliittymään, joka laskee ja esittää tulokset samaiselle näytölle. Kuvassa 4.1 ja 4.2 on esitettynä peruskäyttöliittymä kahdelle eri tuotteen laskentaohjelmille. Ohjelma on tehty Visual Basic 6.0:lla.



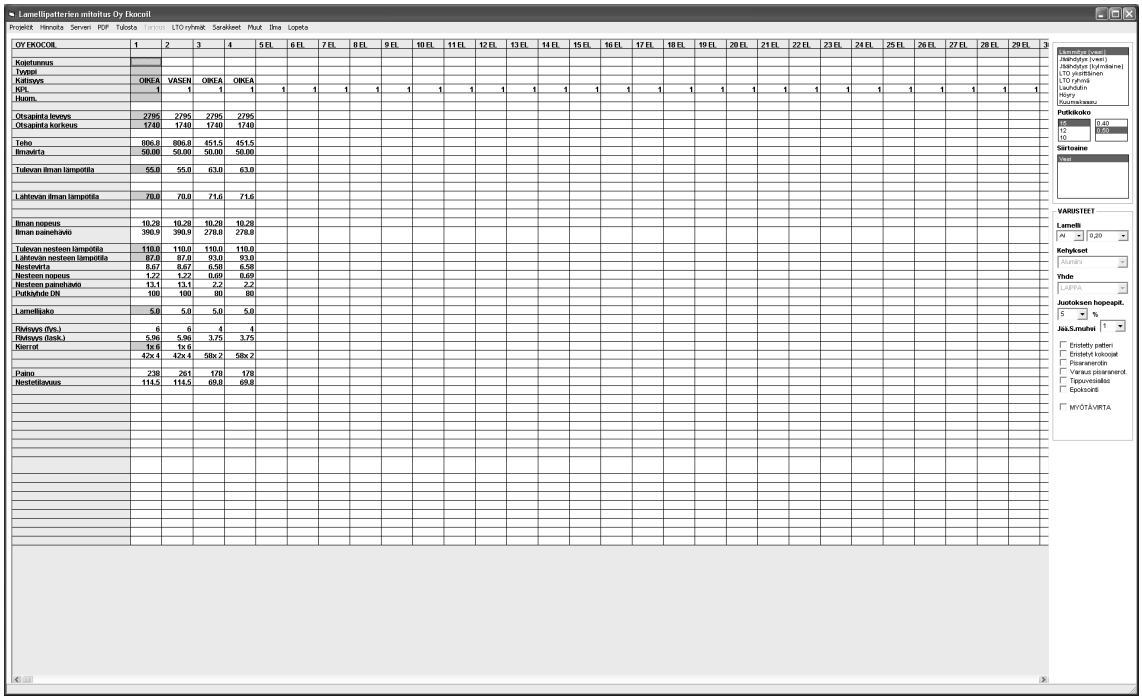
		1 CDSA	2 CDSA	3 CDSA	4 CDSA	5 CDSA
TYYPPI		412-1350-2,5	412-1110-2,5	512-870-2,5	612-630-2,5	713-630-2,5
Ilmavirta	m³/s	5,90	5,03	3,52	2,43	4,25
Lähtevä ilma	°C	33,9	34,6	36,6	39,7	35,5
Lähtevä neste	°C	35,0	35,2	35,1	36,0	35,0
Nestevirta	l/s	0,86	0,89	0,87	0,99	0,86
Painehäviö	kPa	23,1	57,9	17,3	80,2	15,9
Äänitaso Lp	dB(A)	54	49	47	42	44
Putkiyhde	DN	25	25	25	25	25
Reittien lkm		7	5	9	6	9
	kW	2,20	1,52	0,68	0,42	0,63
	A	5,56	3,12	1,68	0,92	1,38
VAPAAJÄÄHDYTYS						
Teho	kW					
Lähtevä ilma	°C					
Lähtevä neste	°C					
Nestevirta	l/s					

Kuva 4.1: Nestejäähdyttimenlaskentaohjelma

Ohjelmasta on olemassa kaksi rinnakkaista versiota. Kaikkein vanhin versio sisältää käyttöliittymänkoodin laskentakoodin seassa. Tätä versiota kutsutaan käyttöliittymälliseksi versioksi.

Toinen versio on niin kutsuttu dll-versio, jossa laskentakoodista on erotettu käyttöliittymän vaatima koodi. Tästä on sen jälkeen käännetty dynaaminen linkitetty kirjasto (dynamic link library dll). Dll:ää käytetään muiden ohjelmien laskentamoduulina. Tämä versio toimii versiona, jota käytetään web-arkkitehtuuriin muuntamiseen, koska käyttöliittymällisenversion ositettavuus on erittäin huono.

Eriytetty laskentamoduuli koostuu useista, monen tuhannen koodirivin, moduuleista. Nämä moduulit ovat teoriassa erillisiä kokonaisuuksia, mutta johtuen ohjelman tavasta käyttää globaaleja muuttujia ja tietorakenteita, moduulit ovat erittäin sidottuja toisiinsa. Riippuvuussuhteiden takia on hyvin vaikeaa ottaa vain muutamaa moduulia ja uudelleen kirjoittaa vain nämä muutama moduuli. Uudelleenkirjoittamisen vaikeuden takia ensimmäisenä muunnos askeleena ei otettu ohjelman täydellistä uudelleen kirjoittamista, koska se olisi kestänyt kauan ja olisi ollut liian riskialtista virheille ja epäonnistumiselle.



Kuva 4.2: Patterilaskentaohjelma

4.2 Projektin suuntaukset

Ensimmäisenä piti päättää miten tätä uutta ohjelmaa lähdettäisiin tekemään. Vaihtoehtoina oli perinteinen työpöytäsovellus ja web-sovellus. Työpöytäsovellus idea hylättiin siitä syystä, että alun suunnittelu vaiheessa idea oli tehdä vain käyttöliittymä uusiksi eikä muokata laskentaosaa mitenkään. Työpöytäversio hylättiin myös sen todennäköisen toteutuskielen takia. Java oli päätöshetkellä todennäköisin toteutuskieli johtuen aikaisemmasta kokemuksesta, ja Javalla oli myös kohtuullisen helppoa tehdä käyttöliittymiä.

Aikaisempien kokemusten mukaan Javalla oli helppo kyllä tehdä suhteellisen yksinkertaisia käyttöliittymiä, mutta monimutkaisempien käyttöliittymien tekeminen meni vaikeaksi ja aikaa vieväksi. Suurin monimutkaisuuden aiheuttaja oli monimutkaisten interaktiivisten taulukoiden tekeminen. Javan taulukoiden piirtäjät ja taulukon solujen piirtäjät alkoivat hajautua niin moneen luokkaan ja tiedostoon, että niitä oli vaikea ylläpitää ja kehittää.

Toinen syy Javan, ja sen mukana työpöytäsovelluksen, hylkäämiseen oli huonot kokemukset dll:n käyttämisestä Javassa. Aikaisemmissa ohjelmissa dll:ien kutsumiseen oli käytetty Java Native Interface (JNI) kirjastoa. Tämä ei kuitenkaan koskaan ollut kauhean vakaata vaan aiheutti useasti ohjelmien kaatumista. JNI:llä toteutettujen testiohjelmien epävakauden takia Java hylättiin.

Päätös oli tehty siirtyä web-maailmaan ja alkaa kehittää laskentaohjelmalle käyttöliittymäsivustoa. Seuraavaksi piti päättää millä web-tekniikalla sivustoa alettaisiin kehittää ja tehtäisiinkö laskenta asiakaspäässä, eli selaimessa, vai palvelinpäässä.

Vaihtoehtoina näyttölogiikan tekemiseen oli Adobe Flash, ja PHP ja JavaScript. Flash:stä ei ollut paljoakaan kokemusta, mutta sillä tiedettiin pystyttävän kehittämään monimutkaisia ja interaktiivisia käyttöliittymiä. Flash:llä oli myös etuna se, että näytöt olisivat toimineet melko suoraan lähes kaikilla alustoilla. Flash olisi myös mahdollistanut asiakaspään laskentaohjelman tekemisen.

Ongelmina Flash:lle pidettiin sen mainetta hyökkäyksille alttiina alustana ja sen huhuttua pikaista kuolemaa HTML 5:en tultua. Lisäongelmana oli Flash kehitystyökalujen maksullisuus ja sulkeutuneisuus. Näiden lisäksi ongelmaksi laskettiin myös ohjelmiston kehittäjän kokemattomuus Flash:stä ja Action Script-ohjelmointikielestä. Jos Flash:llä olisi tehty vain näytöt, olisi palvelinpäänohjelmointi kuitenkin tehty PHP:llä.

Toinen vaihtoehto, PHP ja JavaScript, oli Flash:stä eroten täysin ilmainen ja niille oli paljon ilmaisia, hyviä avoimenkoodin kehitystyökaluja. Ohjelmointi kokemus PHP:stä ja JavaScriptistä oli myös suurempi. Näistä syistä johtuen projekti päätettiin aloittaa

PHP:llä ja JavaScriptillä. PHP:llä pystyttiin myös tekemään palvelinpään ohjelmat, joten laskenta saatiin pidettyä yrityksen omissa palvelimissa.

5 LASKENNAN ONGELMA

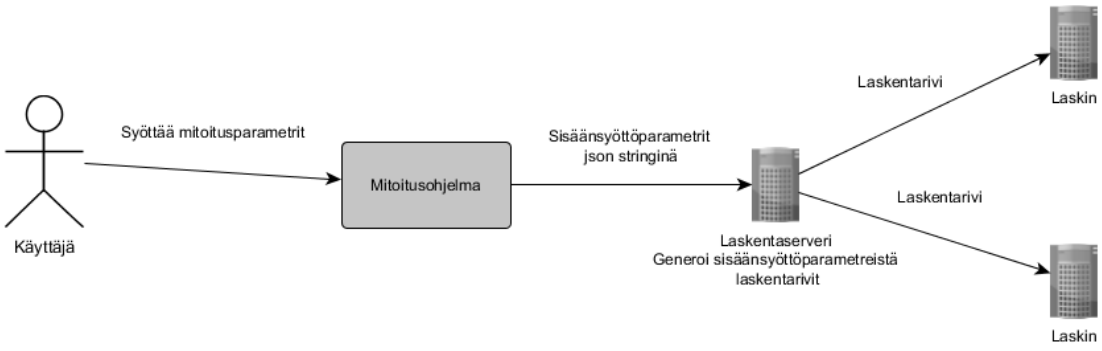
Tämä luku käsittelee ohjelman sisältämän laskennan suurinta ongelmaa, sen hitautta, ja kuinka tätä ongelmaa lähdettiin ratkaisemaan. Luvussa esitellään myös tekniikka, jolla ongelma loppujen lopuksi ratkaistiin.

5.1 Laskennan hitaus

Vanhassa mitoitusohjelmassa käyttäjä syöttää ohjelmalle sisäänsyöttöparametrit, joiden pohjalta ohjelma suorittaa laskennan. Ohjelma ei suorita läheskään niin montaa laskentaa kerrallaan kuin uuden ohjelman haluttiin suorittavan.

Uudessa mitoitusohjelmassa käyttäjä syöttää ohjelmalle sisäänsyöttöparametrit, joista sitten palvelimella generoidaan laskentaohjelmalle annettavat sisääsyöttörivit. Sisäänsyöttörivi sisältää yhden tuotteen laskemiseen tarvittavat arvot. Nämä arvot ovat generoitu käyttäjän antamien arvovälien avulla.

Arvojen generoinnin jälkeen ne menevät yksi rivi kerrallaan laskimelle, joka käyttää saamiaan tuotearvoja ini-tiedostoista arvojen etsintään. Näitä ini-tiedostoista löydettyjä arvoja ohjelma sitten käyttää sisäisissä laskentasilmukoissa laskeakseen mahdollisen kojeen, joka vastaa annettuja sisäänsyöttörivin arvoja. Tuloksena on joko koje tai ei tulosta ollenkaan. Tämä prosessi on kuvattuna kuvassa 5.1.



Kuva 5.1: Laskentaprosessi

Yksittäisen sisäänsyöttörivin laskenta ei kestä kauaa, noin 200–300 millisekuntia riippuen laskinkoneesta ja laskettavasta tuotetyypistä. Mutta sisäänsyöttörivejä yhdelle las-

kennalle tulee vähintään 396. Joten laskenta-ajaksi tulee yli minuutti, joka ei ole hyväksyttävä viive laske-napin painamisen ja tulosten saamisen välillä.

Laskentaa olisi voitu nopeuttaa kiristämällä sisäänsyöttöparametreista generoitujen sisäänsyöttörivien määrää, mutta tämä olisi voinut johtaa siihen, että hyväksyttävillä sisäänsyöttöparametreilla ei olisikaan saatu käypiä vastauksia. Tämä vastausten puuttuminen olisi taas voinut johtaa siihen, että myyjä olisi voinut kuvitella, että ei olisi olemassa kojetta, joka vastaa annettuja parametreja.

Ratkaisuksi nopeusongelmaan suunniteltiin kaksi eri ratkaisuvaihtoehtoa. Ensimmäinen vaihtoehto oli laskentojen etukäteen suorittaminen tietokantaan ja toinen vaihtoehto oli reaaliaikainen laskenta.

5.2 Etukäteen laskenta

Ensimmäinen keino, jolla ohjelmaa päätettiin koettaa nopeuttaa, oli laskea valmiiksi tulosarvoja tietokantaan. Tästä kannasta olisi sitten haettu käyttäjän syöttämiä arvoja vastaavat tulokset ja jos tuloksia ei olisi ollut, olisi suoritettu aikaa vievä laskenta.

Tietokannan nopeuttava vaikutus riippui hyvin pitkälti siitä, että se olisi sisältänyt kaikkein yleisimpien sisäänsyöttöarvojen tulokset. Yleisimpiä sisäänsyöttöarvoja ja niiden ala- ja ylärajoja kerättiin myyjiltä, ketkä tällä hetkellä ovat kaikkein eniten tekemisissä mitoitusohjelman kanssa.

Sisäänsyöttöarvojen ja niiden ala- ja ylärajojen muodostamien välien kautta saatiin ennakkoon laskettavien tulosten määräksi 1 183 604 400 riviä. Tämä rivimäärä oli vain yhden tuotekategorian rivit. Nämä rivit tallennettaisiin MySQL-tietokantaan. Näin muodostettuun tauluun tehtiin indeksointi, joka vastasi normaalia kyselyä, joka tauluun tehtäisiin. Indeksiin listattiin kaikki ne arvot joilla käyttäjä rajaa hakuaan ja ne laitettiin indeksiin siihen järjestykseen kuin ne tulisivat olemaan näytöllä. Tällä indeksillä pyrittiin nopeuttamaan suuresta taulusta hakua.

Tulosarvojen laskenta tietokantatauluun yhdellä koneella olisi kuitenkin kestänyt liian kauan ja kukaan ei myöskään halunnut käsityönä alkaa jakamaan laskenta-arvoja eri koneille. Tähän ongelmaan päätettiin hakea apua hajautetun laskennan hallintaohjelmasta HTCondorista [8].

HTCondorissa yhdestä koneesta tehtiin palvelin, joka jakoi töitä sen verkkoon liitetyille koneille. Liitetyt koneet saivat olla missä tahansa ja mitä koneita tahansa, kunhan ne täyttivät työlle määritellyt ominaisuudet. Tässä tapauksessa vaaditut ominaisuudet olivat laskenta dll:n omistaminen ja Windows-käyttöjärjestelmä. Palvelimella pystyttiin luo-

maan töitä, jotka sisälsivät noin miljoona laskentariviä jokainen. Nämä työt laitettiin työjonoon ja palvelin jakoi ne kaikille laskentakoneille automaattisesti. Palvelin piti myös huolen siitä että, kun laskenta oli valmis, sai laskin uuden työn. Laskentafarmi koostui kaiken kaikkiaan seitsemästä koneesta joista yksi oli uusi Intel i7 kone, ja loput koneet olivat toimistosta käytöstä poistettuja ylimääräisiä koneita.

Tietokannan nopeushyötyä testattiin testikannalla johon luotiin yhden tuoteperheen tulostaulu. Testitauluun generoitiin 52927770 riviä ja siihen myös luotiin edellä kuvattu indeksi. Nopeus taulusta haettaessa ilman indeksi oli kertaluokkaa 413,89 sekuntia ja täydellä järjestetyllä indeksillä 0,78 sekuntia. Vaikka tämä hakunopeus kasvaisi lineaarisesti rivimäärän kasvaessa, olisi täyden taulun haussa kestänyt noin 18 sekuntia. Taulukossa 5.1 on listattuna myös muita testitapauksia ja niillä saatuja hakuaikoja. Osassa testeistä käytettiin *liupia*, joka on parametrinimi liuospitoisuudelle. Liuospitoisuus on yksi laskentaa hyvin rajoittava parametri. Tietokantakoneena toimi virtuaalinen Ubuntu Linux-kone. Koneelle oli annettu 4 cpu:ta ja 8 Gt muistia. Kovalevyt, joille kantaa tallennettiin, oli 10 000 rpm:n sas-levyjä.

Taulukko 5.1: Testikantaan suoritettujen hakujen aikoja

Testi	**Haku aika (s) 52 927 770 rivistä**
Select ilman indeksejä	413,89
Järjestelemätön select indeksillä	4,29
Järjestelty select ilman liupia	6,1
Järjestelemätön select liupilla ja indeksillä	0,86
Järjestelty select liupilla ja indeksillä	0,78
Järjestelty select liupilla, indeksillä ja mahdollisimman isot arvot	5,55

Nopeus, jolla tietokannasta tuloksia saatiin, oli tarpeeksi nopea ja tätä pidettiinkin parhaana vaihtoehtona ohjelman nopeuttamiseen. Ongelma, jota jo aluksi oli pelätty, kuitenkin johti tietokanta ratkaisun hylkäämiseen.

Hieman sen jälkeen, kun tulosten laskenta oli käynnistetty laskentafarmilla, huomattiin laskentamoduulissa virhe. Virhe johti tulosten vääristymään, joten vanhalla laskentamoduulilla laskettuihin tuloksiin ei voitu enää sata prosenttisesti luottaa. Tämä olisi johtanut jo laskettujen rivien poistamiseen ja uudelleen laskentaan.

Ilman suuria lisäpanostuksia laskentafarmiin, tietokannan populointi olisi kestänyt lähes kymmenen kuukautta. Tähän aika-arvioon päästiin laskemalla parhaan koneen laskenta-aikoja. Parhaalla koneella kesti 58 006 249 rivin laskennassa 13 päivää 4 tuntia ja 19 minuuttia. Muut laskentafarmin koneet olivat huomattavasti tätä hitaampia. Ja koska laskentamoduulia ei ole nykyaikaisten testaus menetelmien mukaisesti testattu, ei uusien

virheiden mahdollisuutta voitu poissulkea. Virheiden pelon lisäksi oli tiedossa, että uusia tuotteita ja muutakin tuotekehitystä olisi tapahtumassa lähiaikoina runsaasti. Johtuen uudestaan laskennansuuresta ajasta ei tietokantavaihtoehtoa enää voitu pitää parhaana vaihtoehtona.

5.3 Reaaliaikainen laskenta

Seuraava keino, jolla laskentaa pyrittiin nopeuttamaan, oli jakamalla pieni joukko laskentoja, noin 400 riviä, useammalle koneelle. Tähän ensiksi mietittiin käytettävän HTCondoria, mutta koska laskennat oli tarkoitus tehdä käyttäjän syötteestä, oli HTC:n viive töiden aloituksessa liian suuri, jotta se olisi ollut sopiva työkalu.

Toisena vaihtoehtona oli käyttää niin kutsuttuja message queue palveluita. Message queue palvelussa on jono, tai useampia jonoja, joita erilliset ohjelmat käyttävät toisillensa viestien välittämiseen. Message queuen ero normaaliin viestinvälittämiseen on se, että viestit välitetään kolmannelle osapuolelle. Tämä taas johtaa esimerkiksi siihen, että viestien välittäjien ja vastaanottajien ei tarvitse olla kirjoitettu samalla ohjelmointikielellä, kunhan viesti on sellaisessa muodossa, että kumpikin ohjelma sen ymmärtää. Näistä message queue palveluista mietittiin RabbitMq:ta ja ZeroMq:ta. RabbitMq hylättiin yrityksen aikaisempien sisäisten huonojen kokemusten takia ja koska se olisi vaatinut jälleen yhden palvelimen lisäämistä verkkorakenteeseen. Tämä palvelimen lisääminen olisi lisännyt vikaherkkyyttä ja monimutkaistanut palvelinrakennetta.

ZeroMq on yksinkertainen porttikommunikointi API tai verkko API. Sen avulla voidaan luoda erilaisia palveluita, jotka keskustelevat toisilleen porttien välityksellä. ZeroMq tarjoaa sisäisesti toteutettuja jonoja joihin se tallentaa automaattisesti saamansa viestit. Ohjelmoijan ei siis tarvitse itse useinkaan ottaa suoraa kantaa siihen, kuinka jonot tehdään. [7.]

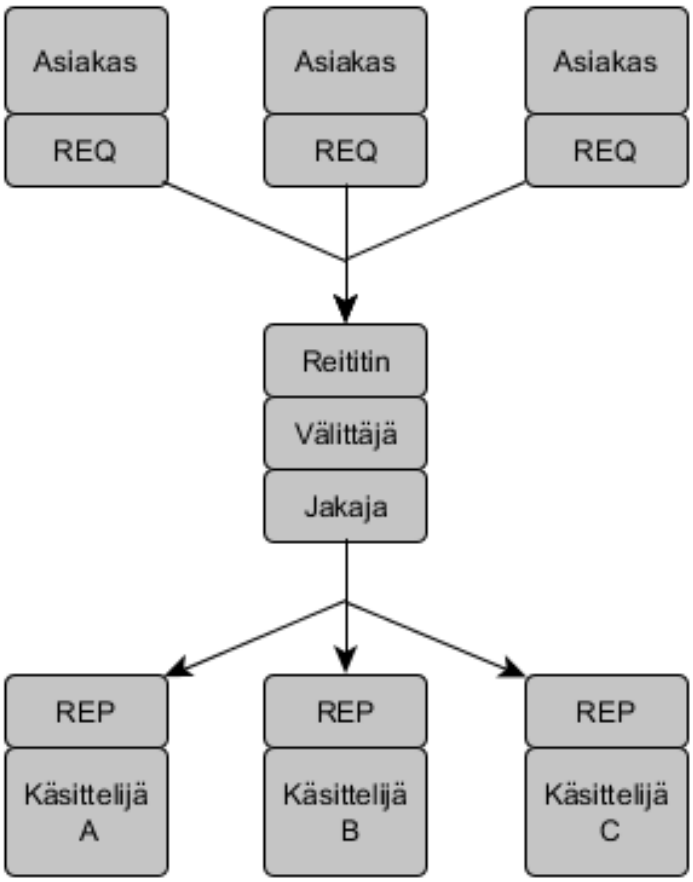
Pääosassa erilaisia arkkitehtuureja suunniteltaessa ZeroMq:lla on useat erilaiset porttityypit. Näitä ovat muun muassa pyyntö (Request), vastaus (Reply) -portit. Nämä kaksi porttia toimivat yksinkertaisimmillaan samalla tavalla, kuin normaalit http-kyselyt. Pyyntöportti lähettää pyynnön vastausportille, joka vastaa kyseiseen pyyntöön. Pyyntöportit ovat synkronisia, eli ne odottavat vastausta pyyntöönsä ennen kuin ne etenevät [7].

Toinen porttipari, jota tultiin käyttämään, on reititin–jakaja-portit (Router, Dealer). Reititinportti vastaanottaa kyselyjä ja luo niistä sisäisesti osoitetaulukon, jonka avulla se voi ohjata vastausviestin takaisin oikeaan osoitteeseen. Välittäjä- tai jakajaportti ei taas ota mitään kantaa viestiin vaan jakaa ne reilusti kaikille kuuntelijoille. Välittäjä myös

jonottaa saamansa vastaukset reilusti. Sekä reititin että välittäjä ovat kummatkin asynkronisia, eli ne eivät odota vastausta ennen uuden viestin käsittelemistä. [7.]

Kuvassa 5.2 esitetään yleisellä tasolla kuormantasausta ZeroMq:lla. Asiakkaat (Client) lähettävät pyyntönsä reitittimeen (Router). Reititin siirtää pyynnön sisäisesti jakajalle (Dealer), joka vuorostaan siirtää pyynnön vapaalle käsittelijälle (Service). Käsittelijä käsittelee pyynnön ja palauttaa tuloksen asiakkaalle ketjun läpi.

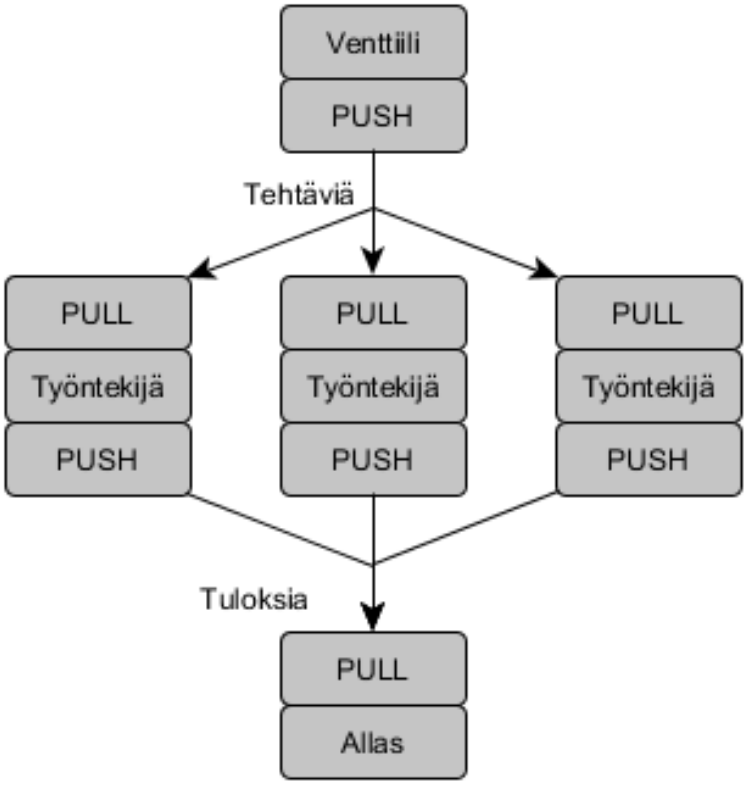
Tällaista arkkitehtuuria käyttämällä asiakkaan ei tarvitse tietää käsittelijästä mitään, vaan tietää ainoastaan reitittimen osoitteen. Myöskään käsittelijät eivät tiedä asiakkaista mitään. Ne tietävät vain jakelijan osoitteen. Näin ollen ainoa kiinteä osa arkkitehtuuria on niin kutsuttu välitin (Proxy), jonka ansiosta palvelua on helppo skaalata [7].



Kuva 5.2: Kuormantasaaja [7, kuva 17]

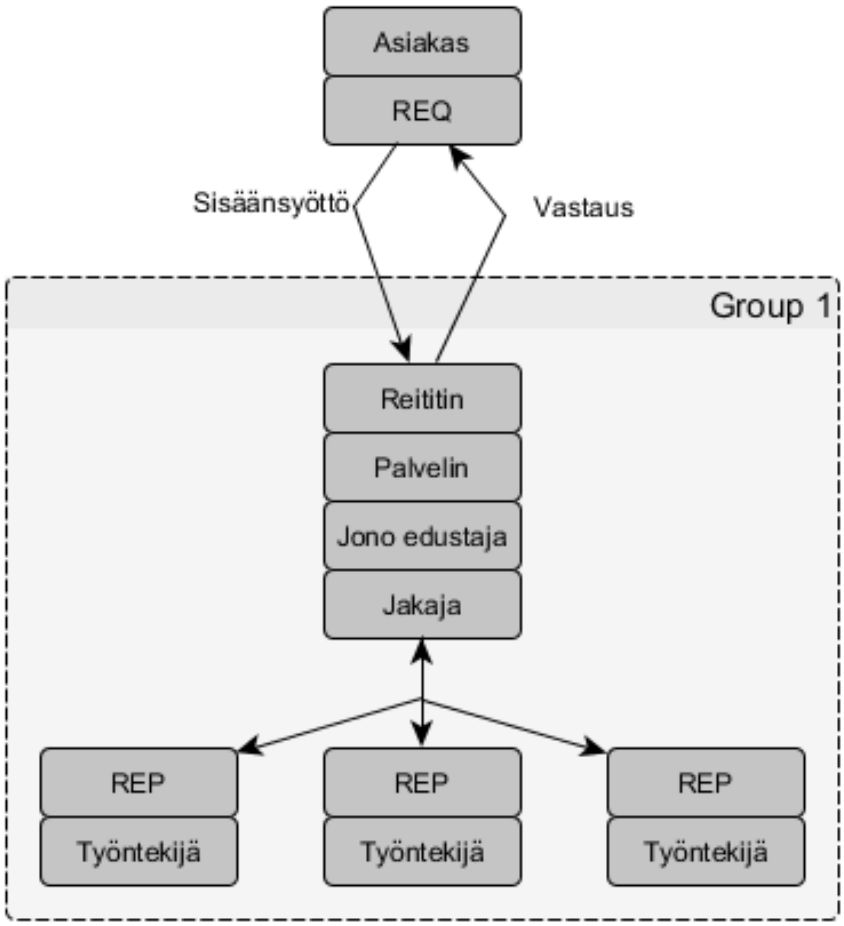
ZeroMq:n avulla pystyy myös tekemään rinnakkaistamista. Kuva 5.3 esittää rinnakkaisenputken arkkitehtuurin. Siinä venttiili (Ventilator) syöttää tehtäviä työntekijöille (Worker), jotka tehtävän tehtyään puskee vastaukset altaaseen (Sink). Tässä arkkiteh-

tuurissa työntekijöitä on helppo lisätä, koska ne rekisteröityvät sekä venttiilille että altaalle. Näin ollen kiinteitä osia arkkitehtuurissa ovat vain venttiili ja allas.



Kuva 5.3: Rinnakkainenputki [7, kuva 5]

ZeroMq:lla pystytään myös tekemään ohjelman sisäistä rinnakkaistamista rinnakkaisen putken lisäksi. Tähän se soveltuu erinomaisen hyvin, koska ZeroMq:n sisäisen toteutuksen ansiosta ohjelman ei tarvitse käyttää muista tavallisista rinnakkaistamiskirjastoista tuttuja lukkoja tai semaforeja. ZeroMq huolehtii säikeiden välisestä kommunikoinnista. [7.] Rinnakkaistamisesta esimerkkinä on kuva 5.4, jossa asiakas lähettää pyynnön palvelimelle. Palvelin sitten jakaa työn sen käynnistämille työntekijöille. ZeroMq pitää huolen sisäisestä tiedonsiirrosta ilman ohjelmoijan tekemiä lukkoja. Tämä arkkitehtuuri on käytännössä sama kuin kuormantasaaja-arkkitehtuuri, mutta tässä työntekijät ja välittäjä ovat laitettu samaan prosessiin.



Kuva 5.4: Säikeistetty serveri [7, kuva 20]

5.4 Rinnakkaistaminen

Laskennannopeudesta tuli yksi tärkeimmistä tavoitteista, ja tätä yritettiin saavuttaa rinnakkaistamalla tapahtuvia laskentoja. Konkreettiseksi tavoitteeksi asetettiin 396 laskentaa viidessä sekunnissa. 396 laskentaan päästiin luomalla laskennoissa käytettävät sisäänsyöttöparametrit myyjiltä saatujen todennäköisten arvovälien avulla. Taulukossa 5.2 on kirjattuna kaikki testitapaukset, joita tehtiin laskentanopeuden testaamisessa. Testikoneena käytettiin Intelin i7-prosessorin sisältävää PC-konetta. Prosessori sisälsi kuusi fyysistä ydintä ja Intelin Hyper Threding ominaisuuden avulla 12 säiettä. Säikeistäminen tehtiin käyttämällä C++ *Boost*-kirjastoa.

Koko-sarakkeessa on laskenta-ajat, kun on käytetty täyttä 396-rivin sisäänsyöttötiedostoa. *Puolikas*-sarakkeessa on laskenta-ajat, kun on käytetty 198-rivistä tiedostoa, ja *neljännes*-sarakkeessa on ajat 99-riviselle tiedostolle. *Poolattu koko*-sarake sisältää laskenta-ajat threadpool:ia käyttäneen ohjelman ajat täydelle 396-rivin tiedostolle.

Threadpool:ssa koko syötetiedosto luettiin muistiin, josta sitten vapaana oleva säie haki itselleen yhden käsiteltävän rivin. Muissa tapauksissa rinnakkaistaminen oli toteutettu lukemalla säikeiden määrän verran rivejä, ja luomalle jokaiselle riville oma säikeensä ja lopuksi odottamalla kaikkien säikeiden valmistumista ja sitten lukemisen jatkamista.

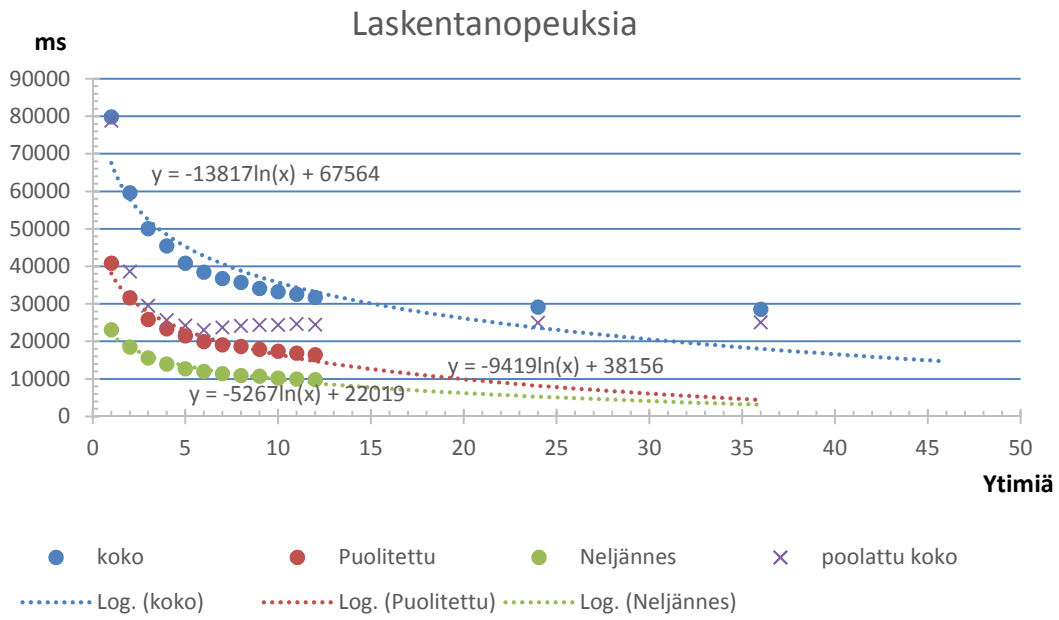
Delta koko-sarake sisältää *koko*-sarakkeen ja *poolattu koko*-sarakkeen välisen aika eron. Tämä laskettiin puhtaasti mielenkiinnosta, jotta nähtäisiin oliko ajoissa eroa kahden erilaisen säikeistämistavan välillä. Näiden kahden säikeistämistavan lisäksi testattiin Intelin Threading Building Blocksia (Tbb) säikeistämiseen. Tbb:llä ajettiin testit kuitenkin vain yhdelle, kuudelle ja kahdelletoista säikeelle. Testejä ei ajettu enempää, koska jo näistä testeistä huomattiin, että laskenta ajat olivat huonompia tai korkeintaan yhtä hyviä kuin threadpoolatut ajat.

Taulukko 5.2: Laskennan nopeusmittauksia

Säikeiden lkm	**Koko tiedosto ms**	**Puolikas tiedosto ms**	**Neljännes tiedosto ms**	**Poolattu koko tiedosto ms**	**∆koko tiedosto ms**
1	79853	40849	23088	78781	1072
2	59654	31671	18501	38659	20995
3	50107	25879	15583	29521	20586
4	45445	23340	13956	25670	19775
5	40872	21520	12734	24244	16628
6	38500	19962	12004	22992	15508
7	36747	19104	11443	23726	13021
8	35742	18640	10912	24176	11566
9	34105	17874	10706	24360	9745
10	33270	17382	10270	24367	8903
11	32539	16899	10011	24661	7878
12	31761	16489	9761	24497	7264
24	29156			24953	4203
36	28548			25086	3462
Min	**28548**	**16489**	**9761**	**22992**	

Kuvassa 5.5 on esitettynä taulukon 5.2 tiedot graafisessa muodossa. Kuvaan on myös piirretty Excelin laskemat trendiviivat. Jos ohjelman ajoajat oikeasti seuraisivat näitä trendiviivoja, tarvittaisiin aikatavoitteeseen pääsemiseen reilusti yli kuusikymmentä ydintä. Kuvaajasta nähtävä tärkein tieto on kuitenkin se, että testikoneessa rinnakkaistamisen hyöty pienenee hyvin nopeasti yli viidellä säikeellä. Tämä nopeuden tasaantuminen sattui osumaan hyvin lähelle suorittimen fyysisten ydinten lukumäärää.

Kaikki testiajot, jotka on merkitty taulukkoon 5.2, on ajettu käyttämällä ohjelman dll-versiota. Niissä ei myöskään ole ylimääräisiä I/O-operaatioita.



Kuva 5.5: Laskentanopeuksia

5.5 Ohjelmankielen muuttaminen

Koska selkeästikään pelkästään sisäänsyöttötiedoston luvun rinnakkaistaminen ja siitä useamman dll:n yhtäaikainen käynnistäminen ei tuonut haluttua nopeutta, päätettiin päästä eroon dll:stä.

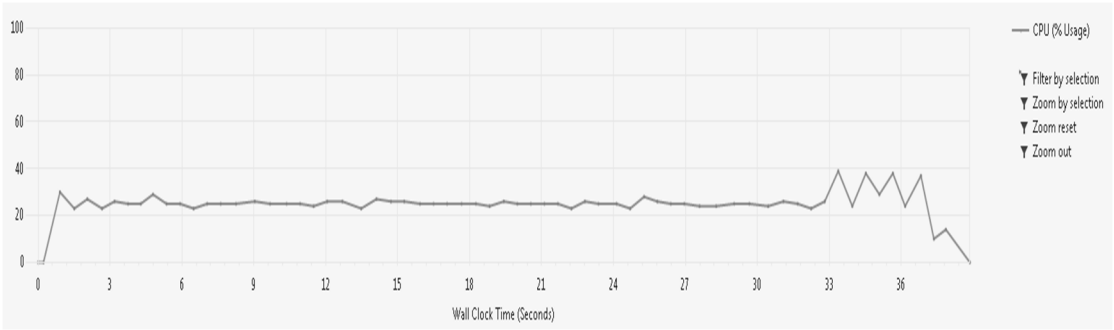
Ohjelmointikielen muuttamiseen oli myös toinen syy. Tämä syy oli halu päästä eroon Windows-käyttöliittymäpakosta. Luonnollisesti, koska C++-kieltä pidetään nopeana, toivottiin myös jo pelkästään ohjelmointikielen vaihtumisen tuovan jonkin verran nopeutta laskentaan vanhaan ohjelmointikieleen verrattuna.

Päätökseen muuttaa ohjelma C++ ohjelmointikielelle vaikutti myös se, että siitä oli jonkin verran kokemusta, sille löytyi automaattinen käännösohjelma ja siitä oli mahdollisuus tehdä PHP-laajennos.

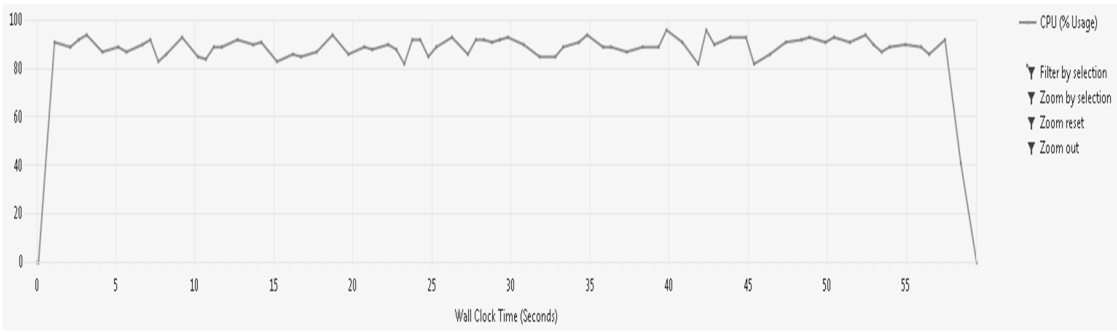
5.5.1 Ohjelmallinen käännös

Automaattinen käännösohjelma pystyi kääntämään Visual Basic 6.0 projektin Visual C++ projektiksi. Ohjelma, jota kääntämiseen käytettiin, oli nimeltään VBto. Ongelma kuitenkin tässä käännöksessä oli se, että kääntäjä ei pyrkinyt eroon Windows-sidoksista käyttämällä standard libraryn (std) tarjoamia vaihtoehtoja, vaan käytti Windowsiin sidottuja kirjastoja. Ohjelma loi myös omia tyyppimuunnos funktioita ja muunsi VB taulukot käyttämällä hyvin vaikeasti ymmärrettäviä ja hallittavia muistioperaatioita. Näistä ongelmista johtuen, vaikka suurin osa kääntämisestä menikin automaattisesti, täytyi käännettyä ohjelmaa käydä läpi vielä käsin.

Kuvat 5.6 ja 5.7 on otettu Visual Studion analyysiohjelmalla. Ne kuvaavat C++ laskin version ja dll laskin version prosessorikuormia. Kuvissa esitetyissä tilanteissa laskentojen peräkkäinen määrä oli eri, mistä johtuen C++-ohjelma näyttäisi valmistuvan nopeammin. Näistä kuvista havaitaan kuitenkin, että käännös C++-kielelle on laskenut prosessorin kuormituksen noin kolmasosaan siitä mitä se oli. Tästä ei voida oikeastaan muuta vielä päätellä, kuin että C++-ohjelma on suuremman osan ajasta odottamassa luku/kirjoitus-operaatioita.



Kuva 5.6: C++ ohjelman prosessori kuorma



Kuva 5.7: Dll-ohjelman prosessori kuorma

Muunnetulla ohjelmalla tehtiin vastaavia nopeustestejä kuin aikaisemmin dll-versiolla. Kaikki testit ajettiin täydellä rivimäärällä, eli 396 rivillä ja yhdellä ytimellä. Ydin rajoi-

tettiin vain yhteen, koska C++-versiota ei voida rinnakkaistaa ohjelman sisäisen rakenteen takia. Testikoneina toimivat kehityskone, joka on virtuaalinen Windows 7 kone, ja sama i7 kone kuin aikaisemminkin. Testien tulokset on koottu taulukoihin 5.3 ja 5.4. Taulukossa 5.3 on laskenta-ajat sekä dll-versiolle, joka toimii tavoitteena, että suoraan C++-kielelle käännetty laskin, ilman mitään optimointeja tai muutoksia. Taulukossa 5.4 on laskenta-aikoja C++-ohjelman eri versioille.

Taulukko 5.3: Nopeusmittauksien kohdearvot C++-ohjelmalla

Aika ms	Huomioita	Testikone
172628	**Dll**	**Testiwin7**
2 252 660	C-käännetty	Testiwin7

Taulukko 5.4: Eri nopeutustapojen aikoja

Huomioita	**C-refaktoroitu (ms)**	**Käännetty optimoidusti (ms)**	**Testikone**
	729861	254179	Testiwin7
Luku muistiin	776499	498282	Testiwin7
Luku merkkijonoon	391 875	160 573	Testiwin7
Luku merkkijonoon		30 747	Calculator01
Ei I/O:ta ollenkaan	129 706	**60253**	Testiwin7
Ei I/O:ta ollenkaan		**5543**	Calculator01

Taulukon 5.3 ensimmäinen arvo on puhtaasti vertailuarvo, johon muita taulukon 5.3 ja 5.4 arvoja verrataan. Se on ajettu dll-versiolla, yhdellä ytimellä ja virtuaalikoneessa. Ensimmäiseksi huomataan, että suoraan VBto-ohjelmalla tehty käännös on huomattavasti hitaampi kuin vertailukohde.

Koska suora käännös oli huomattavasti hitaampi, ohjelma päätettiin refaktoroida käsin. Siitä poistettiin kaikki viittaukset Windows-kirjastoihin ja taulukot muutettiin perus C++ vektoreiksi. Tämä niin kutsuttu C-refaktoroitu versio oli jo huomattavasti nopeampi, mutta silti dll-versioon verrattuna yli neljä kertaa hitaampi.

Seuraava askel oli kokeilla kuinka paljon pelkät C++-kääntäjän optimointivivut auttavat ohjelman nopeuteen. Ohjelma käännettiin käyttämällä Visual Studion kääntäjää, jossa oli optioina /Ox /Ob /Oi. Tämä nopeutti ohjelman puoleen siitä mitä se oli ilman optimoitua käännöstä.

Ohjelma oli saatu huomattavasti nopeammaksi mitä se oli suoraan automaattisen käännön jälkeen ollut, mutta silti se oli edelleen puolitoista kertaa hitaampi kuin alkuperäinen dll-versio. Visual Studiolla ajettujen analyysien perusteella suurimmaksi hitauden aiheuttajaksi havaittiin ohjelman jatkuva tiedostoista lukeminen. Tämä oli jostain syystä huomattavasti hitaampaa C++:lla kuin dll:llä.

Ensiksi päätettiin kokeilla nopeutuisiko ohjelman toiminta, jos se käynnistyessään lukisi kaikki tiedostot avain – arvo taulukkoihin muistiin. Optimointivivuilla käännettynä ohjelma oli itse asiassa kaksi kertaa hitaampi kuin tiedostoista lukeminen.

Seuraavaksi nopeutus yritykseksi valittiin kaikkien tiedostojen muistiin luku, mutta tällä kertaa ne luettiin yhteen pitkään merkkijonoon. Yhden merkkijonon rakenteeseen päädyttiin, koska ilmeisesti avain – arvo taulukosta tuli liian iso ja monimutkainen, joten siitä oli hyvin hidas etsiä haluttua arvoa. Nopeutus tällä metodilla oli tiedostosta lukevaan versioon nähden puolitoistakertainen. Merkkijono lukemisella päästiin jopa hieman nopeammaksi kuin dll-versio. Mutta dll-versiota pystyttiin rinnakkaistamaan ja C++-versiota ei, joten ei tämä nopeus vielä riittänyt.

Viimeinen keino oli poistaa tiedostoista luku kokonaan. Tämä tehtiin pienen apuohjelman avulla, joka luki tiedostot ja muodosti niistä automaattisesti C++-luokan haku operaatioineen. Tällä kovakoodaus keinolla päästiin jo lähelle alkuperäistä viiden sekunnin laskentatavoitetta i7-koneella.

5.5.2 C++-koodin kääntäminen PHP laajennokseksi

Kun C++-koodista oli saatu tarpeeksi nopea laskentaan, piti siitä tehdä PHP-laajennos. PHP-laajennos tehtiin automaattisesti Swig nimisellä työkalulla. Ensin kaikki C++ tiedostot käännettiin objektitiedostoiksi g++ kääntäjällä Linux ympäristössä. Tämän jälkeen Swig:llä luotiin kääritsin C++-luokka käyttämällä Swig:n rajapintatiedostoa. Tämä rajapintatiedosto kuvaa mitä ja miten kääritsin halutaan luoda. Esimerkki tällaisesta tiedostosta on esitettynä listauksessa 5.1

```
%module nestejaahdytin
%include "cpointer.i"
%include "typemaps.i"
%include "std_string.i"
%{
#include "classNestejaahdytin.hpp"
%}

%include "classNestejaahdytin.hpp"
```

Ohjelma 5.1: Nestejaahdytin.i tiedosto

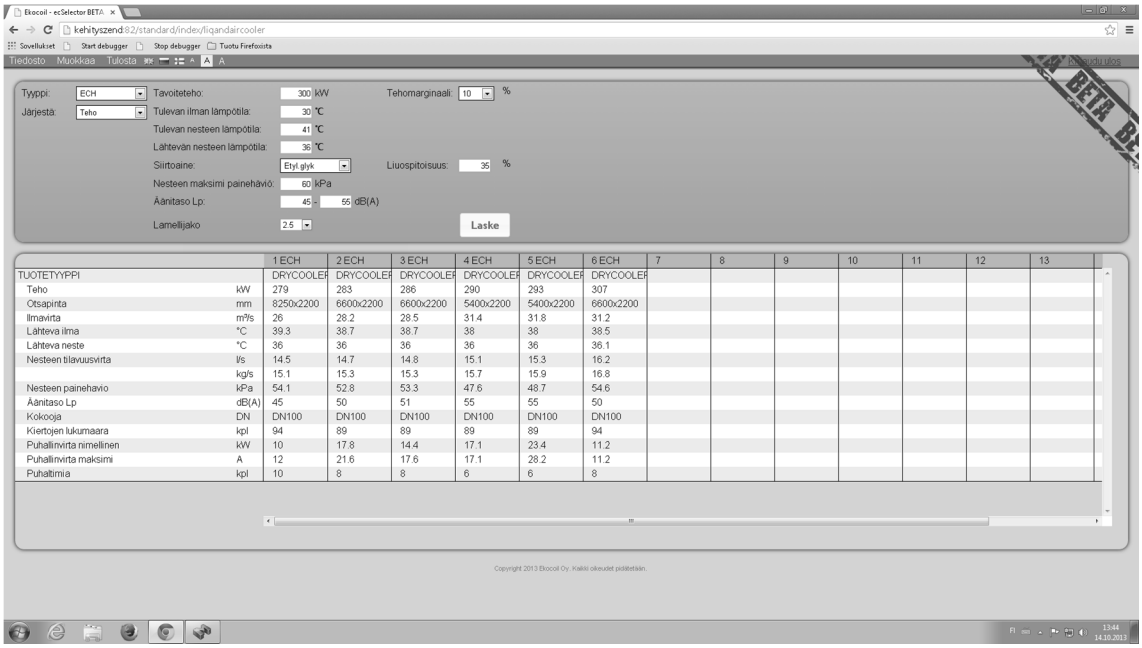
Tämän tiedoston avulla Swig generoi tarvittavat käärimet, tässä tapauksessa nestejäähdytin käärimen. Käärin käännetään g++ kääntäjällä, jonka jälkeen kaikki objektitiedostot voitiin kääntää g++ kääntäjällä Linuxin käyttämäksi *.so* tiedostoksi.

6 TULOKSEKSI SAATU OHJELMA

Tässä luvussa esitellään projektin ja ohjelman tila tällä hetkellä, minkälainen lopullinen verkkoarkkitehtuuri on ja millainen uudistettu mitoitusnäyttö on. Luvussa esitetään myös ajatuksia kuinka projektia tullaan jatkamaan ja minkälainen siitä loppujen lopuksi tulee.

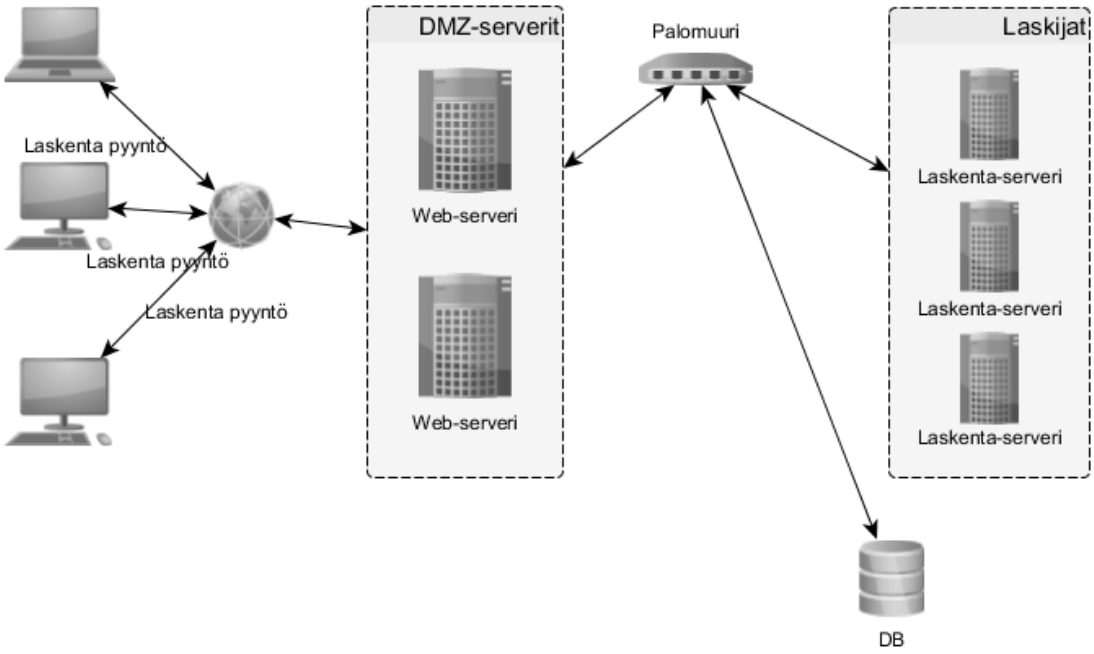
6.1 Ohjelman tila tällä hetkellä

Ohjelman käyttöliittymää on muutettu vahvasti pois Excel tyylisestä taulukkolaskenta näkymästä, modernimman ja selkeämmän näköiseksi. Esimerkki mitoitusohjelman näytöstä on kuvassa 6.1. Tämä näyttö on vakiotuotteen laskentanäyttö, jossa asiakas pystyy mitoittamaan haluamansa tuotteen sisäänsyöttöparametrien avulla.



Kuva 6.1: Käyttöliittymä vakiotuotteen laskennalle

Ohjelma on tällä hetkellä toimivassa kunnossa osalle kaikista tuotteista. Ohjelmiston käyttämässä verkkoarkkitehtuurissa päädyttiin kuvan 6.2 mukaiseen ratkaisuun.

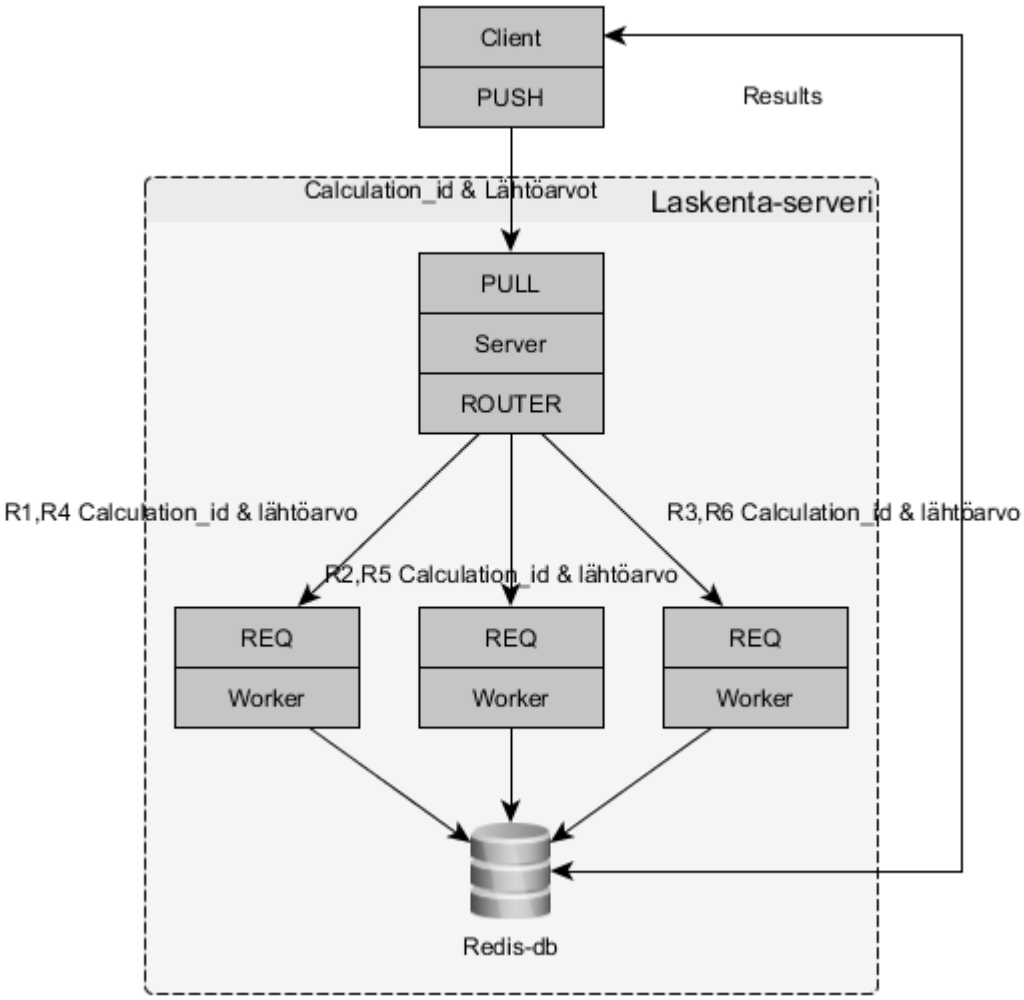


Kuva6.2: Verkkoarkkitehtuuri laskenta ohjelmistolle

Web-palvelin/palvelimet, jotka tarjoavat pelkästään käyttöliittymää on sijoitettu palomuurin ulkopuolelle demilitarisoituun alueeseen (DMZ). Tämä tehtiin yksinkertaisesti siitä syystä, että palomuuriin ei tarvitse avata porttia ulkomaailmalle. Itse laskinpalvelimet on sijoitettu tietokantojen kanssa palomuurin taakse.

Laskenta tarjotaan RESTful tyylisenä rajapintana. Asiakkaille annetaan URL, johon he tekevät POST-pyynnön mikä sisältää sisäänsyöttöparametrit JSON-merkkijonona. Asiakas saa vastaukseksi JSON_merkkijonon, joka sisältää kaikki saadut vastaukset. Koska tällä hetkellä laskentaa tarjotaan vain omille ohjelmistoille, ei laskennan päässä ole käyttäjätunnistusta.

ZeroMq:lla toteutettu laskentapalvelin on toteutettu kuvan 6.3 mukaisella arkkitehtuurilla. Laskentapalvelin koostuu oikeastaan kahdesta osasta: palvelimesta, joka on arkkitehtuurin kiinteä osa ja voi sijaita missä tahansa, ja työntekijöistä, jotka kiinnittyvät palvelimen määritettyyn porttiin ja joille palvelin jakaa työt tasaisesti. Vaatimuksena työtekijä koneille on pääsy Redis-palvelimelle johon ne laittavat laskentojensa tulokset. Tietokannat, joita ohjelmisto käyttää on kuvattu aliluvussa 6.1.2.



Kuva 6.3: Laskentaserverin arkkitehtuuri

Laskentapalvelin koostuu *Palvelin* osasta jossa on asiakkaalle paljastettu *Pull*-portti. Tähän pull-porttiin asiakas lähettää laskentaparametrit. Sisäänsyöttöparametrit lähetetään *Router*-portin kautta palvelimelle rekisteröityneille laskimille. Fyysinen laskentapalvelin voi olla täysin laskimista erillinen kone. Ohjelmassa 6.1 on esitettynä kuinka portit sidotaan zmq:ssa, kun käytetään Node.js:ää toteutuskielenä. Palvelinosa laskentapalvelimesta on toteutettu Node.js:ää käyttämällä ja työntekijät ovat toteutettu PHP:lla, koska C++-laskimesta oli helpompi tehdä PHP-lisäke, kuin Node.js-lisäke, mutta Node.js:ssä oli ajastettujen signaalien lähettämisen tekeminen helpompaa. C++-koodia käyttämällä pyrittiin myös varautumaan mahdollisiin mukauttaviin ylläpitotoimiin.

C++ on laajasti käytetty ohjelmointikieli, joten sen voidaan olettaa pysyvän jatkossakin hyvin samanlaisena ja muuttumattomana. Tämä sama perustelu pätee myös PHP:lle ja JavaScript:lle, jota Node.js käyttää. Käyttämällä vakiintuneita ohjelmointikieliä ja –menetelmiä pystytään helpottamaan mukauttavia ylläpitotoimia.

```
var zmq = require('zmq')
  , PORT = 'tcp://*:12345'
  , PULL_PORT = 'tcp://*:23456'
  , TIME_OUT = 5000 // Aikamillisekuntteina
  , nodemailer = require('nodemailer')
  , mailerTimer = null
, MAIL_TIME = 21600000; // Aika jonka päästä tehdään maili hä-
lytys millisekuntteina

var socket = zmq.socket('router');
var pull = zmq.socket('pull');
```

Ohjelma 6.1: Laskentaserverin porttien sitominen

Ohjelmassa 6.2 on esimerkki laskentapalvelimen töiden vastaanotosta ja lähettämisestä. Kun palvelimelle tulee töitä, se tekee niistä *task*eja jotka se laittaa *workloadiin*. *Workload* on vain yksinkertainen taulukko joka säilöö laskentaolioita. Saadut sisäänsyöttörivit laitetaan taulukkoon, joka on laskentaoliossa.

Töitä lähetetään kymmenen millisekunnin välein, jos on vapaita työntekijöitä ja lähetettäviä töitä. Työn lähetyksessä otetaan aina ensimmäinen työ taulukosta ja ensimmäinen työntekijä työntekijätaulukosta.

```
// Vastaanotetaantyöt
pull.bind(PULL_PORT,function(err){
  if(err)throwerr;

  pull.on('message',function(){
   var calc_id=arguments[0].toString();
   var task=arguments[2].toString();
   if(calc_idinworkload){
     workload[calc_id].tasks.push(task);
   }else{
     tmp=new Calculation(calc_id);
      tmp.tasks.push(task);
      workload[calc_id]=tmp;
   }
  });
});

// Lähetetääntyöt
setInterval(function(){
  if(Object.keys(workload).length>0&&available_workers>0
&&Object.keys(workers).length>0){
    var calc_id=Object.keys(workload)[0];
    var tmp=workload[calc_id];
    var msg=tmp.tasks.shift();

  var key=Object.keys(workers)[0];
```

```
    var worker=workers[key];

    delete workers[key];// Poistetaanvalittuworkkerijonosta
    available_workers--;
    socket.send([worker.id,calc_id,msg]);

    // Tarkastetaan onkotaskissa vielä tehtäviä
    if(tmp.tasks.length<=0){
      delete workload[calc_id];
    }
   }
  },10);
```

Ohjelma 6.2: Esimerkki töiden vastaanotosta ja lähettämisestä

Laskinohjelma ottaa palvelimen lähettämät sisäänsyöttöparametrit yksi kerrallaan vastaan *Req*-portilla. Otettuaan vastaan parametrit, se suorittaa laskennan niiden avulla ja laittaa saamansa tulokset Redis-tietokantaan. Laitettuaan tulokset tietokantaan, se ilmoittaa palvelimelle olevansa valmis vastaanottamaan uudet sisäänsyöttöparametrit.

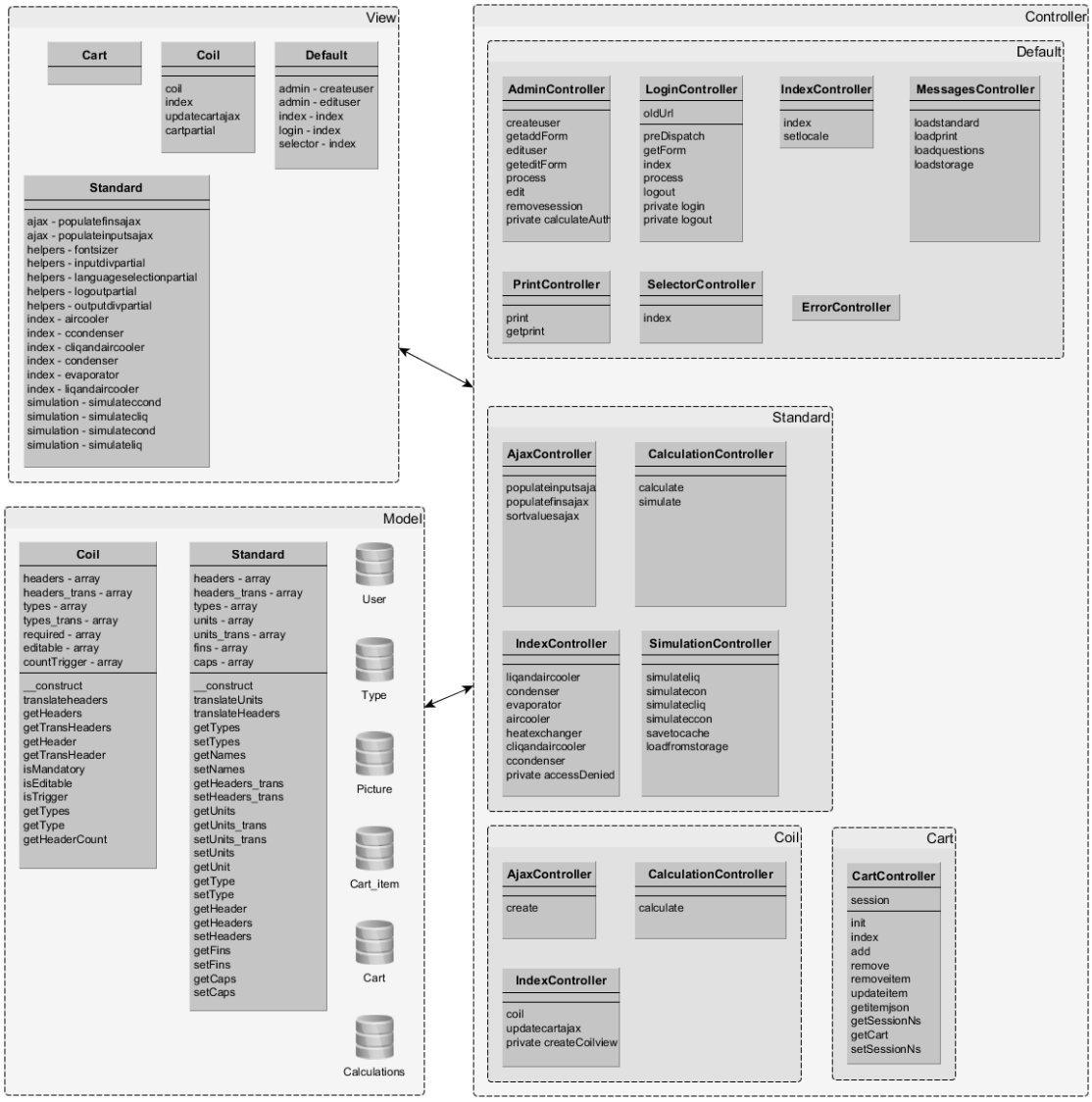
Client on tässä tapauksessa palvelin eikä käyttäjä. Käyttäjä lähettää pyyntönsä palvelimelle, joka luo niistä laskimelle lähetettävät sisäänsyöttöparametririvit. Näiden rivien lähettämisen jälkeen *client* alkaa kyselemään Redis-tietokannalta onko halutulle laskennalle tuloksia. Tätä kyselyä jatketaan kunnes *client* saa tietokannalta oikean määrän vastauksia, tai kunnes se saavuttaa aikakatkaisun.

Ohjelmiston käyttölliittymäosa seuraa MVC-mallia ja on kuvan 6.4 mukainen. MVC-malli otettiin käyttöön, koska se helpottaa erottamaan toiminnallisuuden esityksestä, ja koska käytetty ohjelmointikehys toimii oletuksena kyseisen mallin mukaisesti.

MVC-mallia käyttämällä, ja jakamalla ohjelma mahdollisimman pieniin itsenäisiin osiin, pyrittiin parantamaan jatkossa, sekä ohjelmiston päivitettävyyttä että ylläpidettävyyttä. Vaikka suuri tiedostomäärä saattaa aluksi ylläpitäjälle vaikuttaa vaikealta ymmärtää, niin pienten kokonaisuuksien hallitseminen on kuitenkin helpompaa. Pieniä itsenäisiä osia on myös helpompaa testata.

Normaalista MVC rakenteesta poiketen, malli ei sisällä kaikkea bisneslogiikkaa, vaan on pääsääntöisesti välikerros tietokannan ja ohjelman välillä. Vain *Coil* ja *Standard*–luokat sisältävät suuren määrän toiminnallisuutta. Loput luokat ovat Doctrine 2 orm:n (Object-relational mapper) luomia sidos luokkia, jotka eivät sisällä juurikaan toiminnallisuutta. Ainoastaan *User* ja *Calculations*–luokat sisältävät hieman toiminnallisuutta. Näiden luokkien sisältämät toiminnallisuudet kuitenkin liittyvät tietokannan toiminnallisuuteen ei suoraan ohjelmantoimintaan tai edes businesslogiikkaan.

Kuvassa 6.4 on esitettynä vain PHP-luokat ja vain sellaiset luokat, jotka ovat MVC-mallin mukaiset. Näiden luokkien lisäksi näkymä-osioon kuuluu useita JavaScript tiedostoja, jotka toteuttavat suurimman osan käyttäjän interaktioista ja lisäävät normaalia työpöytäsovellusta imitoivia toimintoja. Lisäksi ohjelmistoon kuuluu kokoelma erilaisia apuluokkia, jotka toteuttavat pieniä aputoiminnallisuuksia, kuten saatavien laskentatulosten järjestelyä, sessionhallintaa, sisäänkirjautumisen ja uloskirjautumisen apufunktioita.



Kuva 6.4: Mitoitusohjelman arkkitehtuuri

6.1.1 Autentikaatio ja autorisaatio

Käyttäjäntunnistaminen toteutettiin itse kirjoitetulla käyttäjätunnistamiskontrollerilla joka periytyy Zend Frameworkin tarjoamasta käyttäjäntunnistamiskontrollerista. Käyttäjän tunnistamisessa oman organisaation sisällä käytetään jo olemassa olevaa Microsoft Active Directory (AD) kantaa, ja ulkopuolisia käyttäjiä varten ohjelmalla on oma käyttäjä-taulua. Kaikki käyttäjät kuitenkin pitää, käyttäjänoikeuksia varten, tunnistaa myös ohjelman omaa käyttäjä-taulua vastaan, koska AD:ssa ei voida käyttäjille määritellä käyttäjätasoja samalla tavalla kuin ohjelmaan ne päätettiin tehdä.

Käyttäjille määritellään käyttäjätasot jokaisen ohjelman ominaisuuden mukaan. Ominaisuuksia tässä tapauksessa on muun muassa jokainen eri tuotekategorian laskeminen ja hintojen näkeminen laskennan tuloksissa. Käyttäjä-taulu sisältää jokaiselle käyttäjälle käyttäjänoikeustason kuvaavan merkkijonon. Tämä merkkijono kuvaa kolmenkymmenen kahden bitin sarjaa.

Jokaiselle ominaisuudelle, joka on haluttu laittaa käyttöoikeuksien taakse, on määritelty vaadittava käyttöoikeustaso. Nämä tasot seuraavat binäärilukujärjestelmällä saatuja arvoja kuten 1, 2, 4, 8, 16, 32, 64, 128… ja niin edelleen.

Käyttäjän tullessa sivulle, jossa on käyttäjänoikeuksia vaativia elementtejä, tarkistetaan käyttäjänoikeusmerkkijono bittioperaatiolla elementin vaatimaan käyttäjänoikeustasoon. Esimerkiksi, jos käyttäjällä on oikeustaso 513, on hän oikeutettu näkemään, sekä tason yksi että tason 512 vaativat elementit (512+1 = 513), mutta hän ei ole oikeutettu näkemään mitään muita tasoja.

Ensimmäiset seitsemän bittiä on varattu tuotekategorioille ja seuraavat seitsemäntoista bittiä on varattu muille oikeuksia vaativille osille, kuten hinnoille ja tuotteenoptioille. Optioiden jälkeinen bitti on varattu kuvaamaan käyttäjän superkäyttäjän oikeuksia. Loput kahdeksan bittiä on tällä hetkellä käyttämättöminä.

6.1.2 Tietokannat

Ohjelmisto käyttää pääasiallisena tietokantanaan Postgre sql tietokantaa. Tämän tietokannan rakenne on esitettynä kuvassa 6.5. Kanta ei sisällä montaakaan eri taulua. Taulut kannassa ovat:

Calculations
Tämä taulu sisältää käyttäjän tekemän laskennan kaikki arvot. Nämä arvot kerätään, kun käyttäjä tulostaa laskentansa. Taulua ei suoraan käytetä mitoitusohjelmassa mitenkään muuten, kuin tallentamisessa, mutta tulostus palvelin Jasper käyttää sitä saadakseen tulosteeseen oikeat arvot.

Calculations	
calc_id	int4
user_id	int4
created	timestamp
modified	timestamp
lahtotietotuoteperhe	int4
lahtotietotuote	int4
lahtotietoliuospitoisuus	float8

Kaikki *Calculations*-taulun arvoalueet on määritelty liitteessä 1.

Cart

Cart-taulu on yhdistävä taulu, taulu itsessään ei sisällä muuta kuin käyttäjän, kenelle kori kuuluu ja kori id:n.

Cart	
cart_id	int4
user_id	int4
created	timestamp
modified	timestamp

Cart_items

Cart_items:ssä yksi rivi kuvaa yhtä tuotetta. Jokainen tuote: kuuluu yhteen koriin, sisältää tuotteen lasketut arvot JSON-merkkijonona ja kuinka monta tuotetta käyttäjä on haluamassa.

Cart_item	
cart_item_id	int4
cart_id	int4
created	timestamp
modified	timestamp
json	text
note	text
qty	int4

Pictures

Pictures-taulu on myös vain tulostukseen tarvittava taulu. Tämä taulu sisältää polut tuotteiden mitoituskuviin, jotka liitetään tulosteisiin.

Picture	
picture_id	int4
path	text

Types

Types-taulu on liimataulu, jolla liitetään tietyn tuotetyyppitekstin omaava tuote sen mittakuvaan. *Size* sarake kertoo kojeen koon yhtiön sisäisesti käyttämällä numerolla. Se ei siis kerro suoraan fyysistä kokoa.

Types	
type_id	int4
picture_id	int4
type_text	text
created	timestamp
modified	timestamp
size	int4

Types_new

Types_new-taulu auttaa muuntamaan uuden tuotekoodin omaavat tuotteet vastaamaan vanhaa tuotekoodia, jotta niille voidaan antaa oikeat mittakuvat. Taulusta etsitään size muuttuja *otpi:n*, *otko:n* ja *type_text*:n avulla. Tämän *size*-muuttujan avulla voidaan etsiä *Types*-taulusta haluttu *kuva*-id.

Types_new	
type_id	int4
type_text	text
created	timestamp
modified	timestamp
size	int4
otpi	int4
otko	int4

Users

Users-taulu sisältää kaikki käyttäjäntiedot, joita käytetään käyttäjäntunnistamisessa ja -oikeuksien hallinnassa. Tämä taulu tulee hyvin suurella todennäköisyydellä kasvamaan johtuen mahdollisista käyttäjäkohtaisista mitoitusohjelma-asetuksista, joita on suunniteltu lisättäväksi tulevaisuudessa.

Salt on jokaiselle käyttäjälle erikseen generoitu satunnais-merkkijono, jota käytetään salasanan kryptaamisen yhteydessä. Tämän suolan lisäksi salasanoihin lisätään ohjelman sisäinen toinen suola. *Sessio_id* pitää sisällään merkkijonon, joka sisäänkirjautumisessa muuttuvan istunnon tunnisteen. Tätä käytetään tunnistamaan onko kyseinen käyttäjä jo kirjautuneena toisesta selaimesta palveluun. Jos käyttäjä on jo sisään kirjautuneena, otetaan uusi istuntotunniste kantaan ja vanhan istunnon omistava selain kirjataan ulos palvelusta.

Users	
user_id	int4
created	timestamp
modified	timestamp
first_name	varchar(255)
surname	varchar(255)
salt	varchar(50)
username	varchar(255)
password	varchar(255)
authlevel	varchar(255)
sessio_id	varchar(255)

Redis tietokannat

Redis on täysin palvelimen muistissa toimiva kehittynyt avain – arvo tietokanta [13]. Redis ei kuitenkaan ole vain puhdas avain – arvo tietokanta vaan sinne pystyy myös tallentamaan listoja, joukkoja, järjestettyjä joukkoja ja pelkkiä merkkijonoja.

Uudessa mitoitusohjelmassa Redis:ä käytetään laskentojen kätkönä, jonne jokainen laskin asettaa laskemansa tulokset, ja josta sitten haetaan tulokset laskenta-id:n perusteella. Laskentojen tulokset, jotka ovat JSON muotoinen merkkijono, asetetaan listaan, jonka avaimena toimii laskenta-id.

```
EkocoilLas1388914153
     {Vastaus 1},
     {Vastaus 2},
     {Vastaus 3},
     {Vastaus 4},
```

Ohjelma 6.3: Laskennantulosten tallennus Redis tietokantaan

Toinen käyttökohde Redis:lle on jatkossa muuntaa ini-tiedostot tietokantakoneen muistissa oleviksi objekteiksi. Nämä objektit luetaan MongoDb-tietokannasta Redis:iin apuohjelman avulla. Tämä lukeminen suoritetaan kerran päivässä, jotta voidaan olla varmoja, että muistissa olevassa tietokannassa on mahdollisimman uudet ja ajan tasalla olevat tiedot.

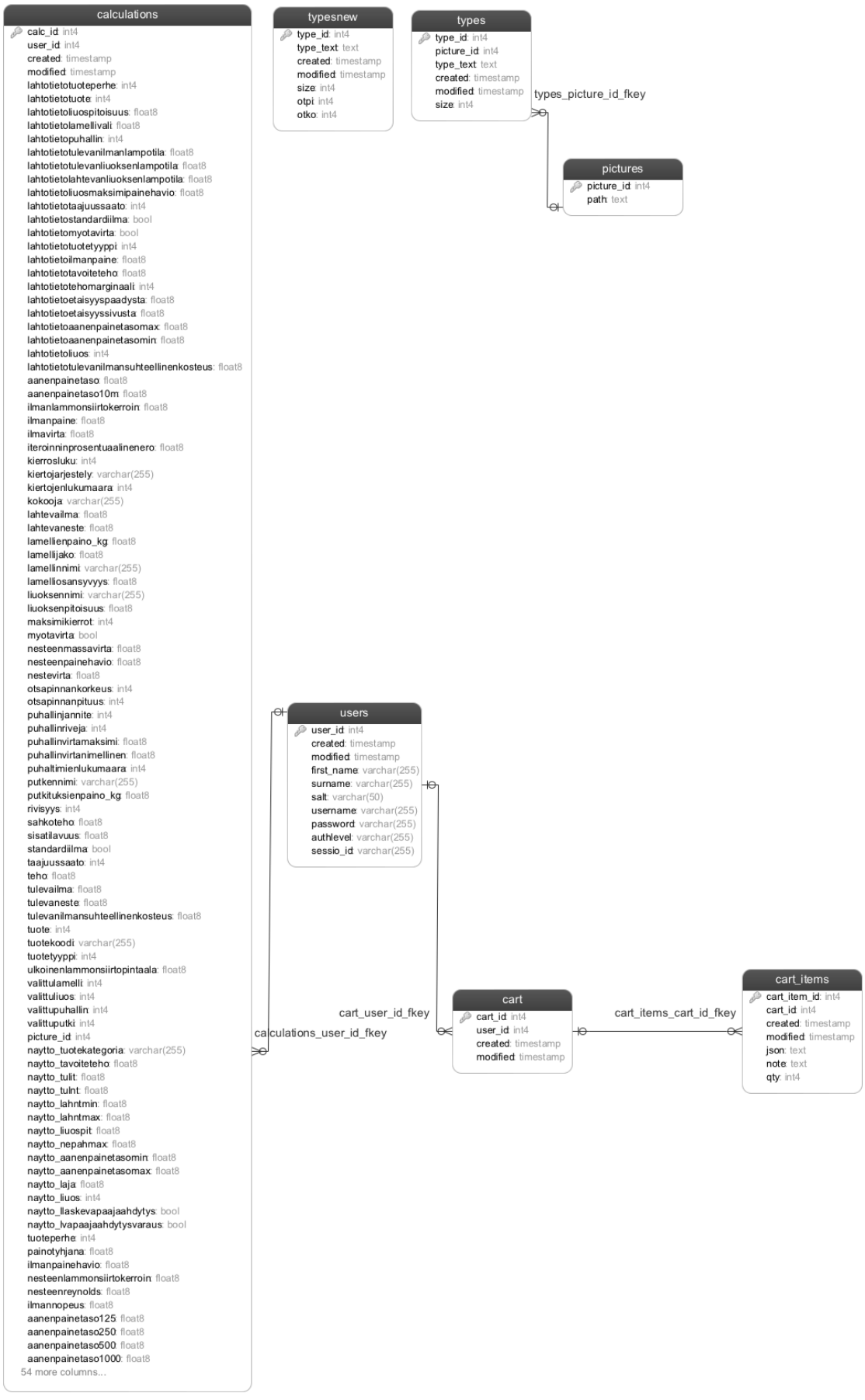
Syy miksi ini-tiedostot siirretään MongoDb tietokantaan eikä Postgre tietokantaan on ini-tiedostojen rakenne. Ne ovat jaettu useaan alaotsikkoon, joista jokaisen alla on yksi tai useampi avain – arvo pari. Lyhyt esimerkki ini-tiedoston rakenteesta on esitetty listauksessa 6.4. MongoDb käyttää JSON muotoisia merkkijonoja joihin se tallentaa tiedot. Sille ei tarvitse määritellä tietokantatauluskeemaa. Tämä skeeman puuttuminen sallii jokaisesta ini-tiedostosta helposti luotavan JSON merkkijonon.

```
'tuoteperhetiedot''''''''''''''''''''''''''''''''''''''''''
[ECH]
Tuoteperhetunnus=EC
TuoteperhetunnusUusi=ECH
Puhallin1=103
Puhallin2=105

[ECH Nestejaahdytin]
Lamellivali1=2
Lamellivali2=2.3
Tuotetyyppitunnus=G
TuotetyyppitunnusUusi=DRYCOOLER
```

Ohjelma 6.4: Ote ECH-tuotekategorian ini-tiedostosta

Käyttämällä tietokantoja myös itse laskentaohjelman asetustiedostoissa helpotetaan näiden tietojen päivittämistä ja lisäämistä. Tietokantaan on helppo tehdä yksinkertainen käyttöliittymä, jolla tietoja pystyy päivittämään ja lisäämään kuka tahansa ilman ohjelmointiosaamista. Tämä taas helpottaa osaltaan ohjelman tulevaa ylläpitoa ja kehitystä.



Kuva 6.5: Mitoitusohjelman käyttämä Postgre-tietokanta

6.2 Projektin jatko

Projekti jatkuu lisäämällä uusia tuotekategorioita, joita sillä voidaan mitoittaa. Näille kategorioille täytyy tehdä omat näyttönsä ja niiden laskentamoduulit täytyy kääntää C++:lle ja siitä vielä PHP-moduuliksi. Nykyisiä mitoitusnäyttöjä on harkittu jo muutettavaksi vielä yksinkertaisemmiksi käyttää, muuttamalla ne kuvapohjaisiksi. Tällöin niissä sisäänsyöttöparametrit annettaisiin niin, että kohdat joihin arvot kirjoitetaan, olisivat esitettynä tuotteenkuvan päällä niillä kohdilla, jotka kuvaavat kyseistä ominaisuutta parhaiten.

Laskentaa on tarkoitus alkaa tarjoamaan myös ulkopuolisille asiakkaille, liitettäväksi heidän omiin mitoitusohjelmiinsa. Tässä tapauksessa laskentaan on myös lisättävä käyttäjien tunnistus ja myös käyttäjänoikeuksienhallinta.

Mitoitusohjelma on myös tarkoitus yhdistää suoraan tuotannonhallinta-ohjelmistoihin verkkokaupoissa käytetyn kauppakorin tapaisella ominaisuudella. Mitoitusohjelman käyttäjä tekisi laskentansa aivan normaaliin tapaan, mutta hänen ei tarvitsisi enää tulostaa laskentaansa ja lähettää sitä sähköpostilla myyjälle. Käyttäjä voisi suoraan siirtää valitsemansa laskennan kauppakoriin ja sitä kautta lähettää tarjouspyynnön myyjille käsiteltäväksi.

Laskentapalvelimien arkkitehtuuria on myös mietitty voitaisiinko sitä yksinkertaistaa. Yksinkertaistamisen lisäksi laskentapalvelimien klusterointia ja laajempaa hajauttamista on alettu miettimään. Yksinkertaistamista pyritään ensisijaisesti tekemään muuntamalla koko laskentaketju käyttämään yhtä ja samaa ohjelmointikieltä. Ohjelmointikielen yhtenäistämisen lisäksi tavoitteena on saada laskentapalvelimesta yksi palvelinkoneelle asennettava paketti, jotta se olisi helpompi ja nopeampi asentaa uusiin koneisiin.

7 JOHTOPÄÄTELMÄT

Ohjelman teossa käytetty ohjelmistokehys Zend Framework versio 1 oli suurena apuna ohjelmiston koodin loogisen rakenteen ylläpitämisessä. Vaikka kehys nimellisesti onkin MVC-kehys, pystyy sillä kuitenkin liian helposti tekemään huonoa MVC-koodia. Tämä on tietenkin myös kiinni ohjelmiston tekijöiden taidoista ja osaamisesta, mutta kehys olisi voinut hieman tarkemmin määritellä kaikkien kolmen (MVC) osan rajat.

Jos projekti aloitettaisiin vasta nyt, ei Zend Framework varmaankaan tulisi valituksi käytettäväksi ohjelmistokehykseksi, koska siitä puuttuu joitakin hyvin oleellisia asioita joita ohjelmistossa käytettiin, kuten sisäänrakennettu ORM. Näiden puutteiden lisäksi kehys on osoittautunut yllättävän laajaksi huonossa mielessä, eli siinä on liikaa turhaa asetusten ja konfiguraatioiden tekemistä. Tämän hetkisen kokemuksen ja tiedon perusteella varmaankin valittavaksi kehykseksi tulisi Laravel 4 [10], josta löytyy muun muassa valmiina ORM ja sen avulla hallittava käyttäjäntunnistaminen ja -oikeuksienhallinta. Myös ohjelmiston rakenteesta tulee selkeämpi ja hallittavampi, kuin Zend Framework:llä.

Suurin syy miksi Zend Framework valittiin, vaikka se ei välttämättä ollut edes projektin alkuvaiheessa paras mahdollinen kehys projektiin, oli yksinkertaisesti vaatimukset nopeasta tuotantokoodin aikaansaamisesta. Koska ohjelmiston kaikkia vaatimuksia ei saatu koottua ja mietittyä ennen ohjelmoinnin aloittamista, ei kehyksen ominaisuuksia tai niiden puutteita heti huomattu. Tämä johti tilanteeseen, jossa koodia oli jo tehty kehyksen ehdoilla niin paljon, että kehystä ei voitu enää vaihtaa kun puutteita huomattiin.

Palvelinpään toteutuksella, vaikka ensisilmäyksellä näyttääkin monimutkaiselta, saavutettiin sille asetetut tavoitteen. Tavoitteet saavutettiin lähinnä ZeroMQ:n avulla saavutetun skaalautuvuuden takia. ZMQ-laskimen toteutuksen ansiosta arkkitehtuurin kiinteän osa saatiin rajoitettua yhteen ohjelmaan ja senkin saisi skaalattua ja rinnakkaistettua pienillä lisäyksillä ja muutoksilla. Itse laskimia voi teoriassa olla niin monta kuin halutaan palvelimia rakentaa. Vaikka itse laskinmoduulia ei saatukaan sisäisesti rinnakkaistettua, riittää se, että jokainen laskenta saadaan suoritettua haluttaessa rinnakkaisesti. Tällä ratkaisulla teoriassa, jos laskimia rakennettaisiin yhtä monta kuin laskentarivejä on maksimissaan, saataisiin laskenta-aika tiputettua lähes yhden rivin laskennan aikaan. Tämä olisi luokkaa 200 – 300 millisekuntia. Tietenkään tällainen laskimien lisääminen ei ole taloudellisesti järkevää, ja jo nykyisellä yhdellä fyysisellä laskinkoneella, jossa on kuusi fyysistä ydintä, päästään alle viiden sekunnin tavoiteaikaan.

8 YHTEENVETO

Projektin tavoitteena oli muuntaa vanha Visual Basic 6.0-ohjelma vastaamaan nykyaikaisiin käyttäjä- ja businessvaatimuksiin. Tämä tehtiin muuntamalla ennen pelkkänä työpöytäsovelluksena toiminut ohjelma Internetissä toimivaksi web-sovellukseksi.

Aluksi piti päättää minkälaisia verkko- ja web-tekniikoita sovelluksen tekemiseen käytetään. Tässä päädyttiin käyttämään PHP-koodikieltä palvelinpään ohjelmointiin ja jQuery JavaScript-kirjastoa asiakaspään ohjelmointiin. Palvelinpäässä PHP:tä käytetään sivujen rakentamiseen ja tietokantayhteyksien ylläpitoon. Laskennan kuormantasaus päädyttiin toteuttamaan ZeroMQ verkko API:a käyttämällä. Tämä kuormantasaus toteutettiin yhdistelemällä Node.js:llä tehtyä ZeroMQ-palvelinta ja PHP:llä tehtyjä laskennan toteuttajia.

Tietokantoina, joita mitoitusohjelma käyttää, käytetään Postgre-tietokantaa Doctrine 2.0 ORM:n avulla ja Redis-tietokantaa. Postgre-tietokanta toimii sovelluksen pääasiallisena tietokantana ja Redis-tietokanta toimii vain väliaikaisena tallennuspaikkana suoritetuille laskennoille, ennen kuin ne lähetetään takaisin käyttäjälle.

Työläimmäksi osaksi ohjelman muuntamista tuli laskennan saaminen tarpeeksi nopeaksi. Ohjelman nopeuttamiseen käytettiin paljon aikaa ja useampaa eri tekniikkaa tavoitteen saavuttamiseksi kokeiltiin.

Ensimmäinen nopeutusyritys oli tietokantaan valmiiksi lasketut laskennat. Laskentaa suoritettiin laskentafarmilla, joka koostui useammasta koneesta joita ylläpiti HTCondor. Tämä nopeutusmenetelmä kaatui laskentamoduulista löydettyyn virheeseen, joka aiheutti jo laskettujen arvojen hylkäämiseen. Aina, jos lasketut arvot olisi pitänyt uudelleen laskea, olisi uudelleen laskentaan mennyt yhden tuotekategorian kohdalla yli kymmenen kuukautta. Tämä oli liian kauan.

Seuraava keino, jota yritettiin, oli laskentaohjelman laskennan dll-version rinnakkaistaminen. Rinnakkaistamiseen käytettiin C++:n *Boost* kirjastoa ja *ThreadPool*-mallia. Rinnakkaistamisella laskenta saatiin nopeammaksi, mutta testien kautta todettiin, että haluttuun tavoiteaikaan pääsemiseen olisi laskimia pitänyt saada yli kuudenkymmenen ytimen edestä. Tämä ytimien määrä oli liian suuri, jotta tällaista rinnakkaistamista olisi voitu harkita ratkaisuksi nopeusongelmaan.

Viimeisenä ratkaisuna nopeusongelmaan kokeiltiin laskimen koodikielen vaihtamista Visual Basicista C++-kieleen. Kieli käännettiin automaattisella muunnosohjelmalla, jonka jälkeen koodi käytiin vielä käsin läpi, jotta päästiin eroon turhista tyyppimuunnoksista ja Windows kirjastojen käytöstä.

Käännetty koodi ei suoraan pärjännyt dll-version nopeudelle. Hitauden aiheuttajaksi todettiin I/O-operaatiot, joten niistä hankkiuduttiin eroon kovakoodaamalla tarvittavat tiedostot C++-luokiksi. Ilman I/O-operaatioita, käännetty koodi olikin jo tarpeeksi nopea. Koodikielen muuntamisen lisäksi nopeutta saatiin lisäämällä verkkoarkkitehtuuriin kuorman tasaus ZeroMQ:lla. Tämän avulla laskimia voidaan lisätä helposti, jos käyttäjämäärät kasvavat liian suuriksi nykyiselle laskennalle, jotta pysyttäisiin edelleen tavoite laskenta-ajan alla.

Tällä hetkellä ohjelma on sisäisessä testauksessa ja sitä se on tarkoitus julkaista asiakkaille kesän 2014 aikana. Tämän jälkeen projektin on tarkoitus jatkua uusien mitoitusohjelmien siirtämisellä samaan arkkitehtuuriin.

LÄHTEET

[1] Aversano, L., Canfora, A., Cimitile, A., De Lucia, A. Migrating Legacy Systems to the Web: an Experience Report. Fifth European Conference on Software Maintenance and Reengineering, 2001. March 14 – 16, 2001. Lisbon. pp. 148-157.

[2] Dhingra, S. 2013. REST vs. SOAP: How to choose the best Web service [WWW]. [Viitattu 12.5.2014]. Saatavissa: http://searchsoa.techtarget.com/tip/REST-vs-SOAP-How-to-choose-the-best-Web-service.

[3] Endrei, M., Ang, J., Arsanjani, A., Chua, S., Comte, P., Krogdahl, P., Luo, M., Newling, T. Patterns: Service-Oriented Architecture and Web Services. April 2004. IBM. 370 s.

[4] Fielding, T., R. Architectural Styles and the Design of Network-based Software Architectures. Dissertation. Irvine 2000. University of California, Information and Computer Science. 152 p.

[5] Haikala, I., Järvinen, H.M. 2004. Käyttöjärjestelmät. Jyväskylä, Talentum. 246 s.

[6] Harsu, M. 2003. Ohjelmien ylläpito ja uudistaminen. Helsinki, Talentum. 292 s.

[7] Hintjens, P. 0MQ – The Guide [WWW]. [Viitattu 16.11.2013]. Saatavissa: http://zguide.zeromq.org/page:all.

[8] HTCondor. [WWW]. [Viitattu 7.5.2014]. Saatavissa: http://research.cs.wisc.edu/htcondor/description.html

[9] jQuery. Test Swarm Wiki [WWW]. [Viitattu 15.12.2013]. Saatavissa: https://github.com/jquery/testswarm/wiki.

[10] Laravel 4. [WWW]. [Viitattu 7.5.2014]. Saatavissa: http://laravel.com/

[11] Microsoft. Model-View-Controller [WWW]. [Viitattu 23.11.2013]. Saatavissa: http://msdn.microsoft.com/en-us/library/ff649643.aspx.

[12] Mueller, J. 2013. Understanding SOAP and REST Basics [WWW]. [Viitattu 12.5.2014]. Saatavissa: http://blog.smartbear.com/apis/understanding-soap-and-rest-basics/.

[13] Redis. Introduction to Redis [WWW]. [Viitattu 5.1.2014]. Saatavissa: http://redis.io/topics/introduction.

[14] RFC-6455. The Websocket Protocol. Internet Engineering Task Force (IETF). December 2011. 69 s.

[15] Selenium Project. Selenium RC [WWW]. [Viitattu 15.12.2013]. Saatavissa: http://www.seleniumhq.org/docs/05_selenium_rc.jsp.

[16] Sneed, H.M., Integrating legacy software into a service oriented architecture, Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on. March 22-24, 2006. pp. 3-14.

[17] Victoria, P., Nickerson, B. G. Displaying Real-Time Data with WebSocket. Internet Computing, IEEE 16(2012)4, pp 45-53.

[18] Wikipedia. WebSocket Browser Support [WWW]. [Viitattu 23.11.2013]. Saatavissa: http://en.wikipedia.org/wiki/WebSocket#Browser_support.

[19] Wikipedia. WSDL [WWW]. [Viitattu 26.8.2013]. Saatavissa: http://fi.wikipedia.org/wiki/WSDL.

[20] W3C. XMLHttpRequest W3C Working Draft 6 December 2012 [WWW]. [Viitattu 23.11.2013]. Saatavissa: http://www.w3.org/TR/XMLHttpRequest/.

[21] Zend. Zend Framework & MVC Introduction [WWW]. [Viitattu 30.11.2013]. Saatavissa: http://framework.zend.com/manual/1.12/en/learning.quickstart.intro.html.

Liite 1: Calculations-taulu

Field	Type	AllowNull	Default Value
calc_id	int4	No	nextval('calculations_calc_id_seq'::regclass)
user_id	int4	Yes	
created	timestamp	No	
modified	timestamp	Yes	NULL::timestampwithouttimezone
lahtotietotuoteperhe	int4	Yes	
lahtotietotuote	int4	Yes	
lahtotietoliuospitoisuus	float8	Yes	
lahtotietolamellivali	float8	Yes	
lahtotietopuhallin	int4	Yes	
lahtotietotulevanilmanlampotila	float8	Yes	
lahtotietotulevanliuoksenlampotila	float8	Yes	
lahtotietolahtevanliuoksenlampotila	float8	Yes	
lahtotietoliuosmaksimipainehavio	float8	Yes	
lahtotietotaajuussaato	int4	Yes	
lahtotietostandardiilma	bool	Yes	
lahtotietomyotavirta	bool	Yes	
lahtotietotuotetyyppi	int4	Yes	
lahtotietoilmanpaine	float8	Yes	
lahtotietotavoiteteho	float8	Yes	
lahtotietotehomarginaali	int4	Yes	
lahtotietoetaisyyspaadysta	float8	Yes	
lahtotietoetaisyyssivusta	float8	Yes	
lahtotietoaanenpainetasomax	float8	Yes	
lahtotietoaanenpainetasomin	float8	Yes	
lahtotietoliuos	int4	Yes	

lahtotietotulevanilmansuhteelli-nenkosteus	float8	Yes	
aanenpainetaso	float8	Yes	
aanenpainetaso10m	float8	Yes	
ilmanlammonsiirtokerroin	float8	Yes	
ilmanpaine	float8	Yes	
ilmavirta	float8	Yes	
iteroinninprosentuaalinenero	float8	Yes	
kierrosluku	int4	Yes	
kiertojarjestely	var-char(255)	Yes	NULL::charactervarying
kiertojenlukumaara	int4	Yes	
kokooja	var-char(255)	Yes	NULL::charactervarying
lahtevailma	float8	Yes	
lahtevaneste	float8	Yes	
lamellienpaino_kg	float8	Yes	
lamellijako	float8	Yes	
lamellinnimi	var-char(255)	Yes	NULL::charactervarying
lamelliosansyvyys	float8	Yes	
liuoksennimi	var-char(255)	Yes	NULL::charactervarying
liuoksenpitoisuus	float8	Yes	
maksimikierrot	int4	Yes	
myotavirta	bool	Yes	
nesteenmassavirta	float8	Yes	
nesteenpainehavio	float8	Yes	
nestevirta	float8	Yes	
otsapinnankorkeus	int4	Yes	
otsapinnanpituus	int4	Yes	
puhallinjannite	int4	Yes	
puhallinriveja	int4	Yes	
puhallinvirtamaksimi	float8	Yes	

puhallinvirtanimellinen	float8	Yes	
puhaltimienlukumaara	int4	Yes	
putkennimi	var-char(255)	Yes	NULL::charactervarying
putkituksienpaino_kg	float8	Yes	
rivisyys	int4	Yes	
sahkoteho	float8	Yes	
sisatilavuus	float8	Yes	
standardiilma	bool	Yes	
taajuussaato	int4	Yes	
teho	float8	Yes	
tulevailma	float8	Yes	
tulevaneste	float8	Yes	
tulevanilmansuhteellinenkosteus	float8	Yes	
tuote	int4	Yes	
tuotekoodi	var-char(255)	Yes	NULL::charactervarying
tuotetyyppi	int4	Yes	
ulkoinenlammonsiirtopintaala	float8	Yes	
valittulamelli	int4	Yes	
valittuliuos	int4	Yes	
valittupuhallin	int4	Yes	
valittuputki	int4	Yes	
picture_id	int4	No	
naytto_tuotekategoria	var-char(255)	Yes	NULL::charactervarying
naytto_tavoiteteho	float8	Yes	
naytto_tulit	float8	Yes	
naytto_tulnt	float8	Yes	
naytto_lahntmin	float8	Yes	
naytto_lahntmax	float8	Yes	
naytto_liuospit	float8	Yes	
naytto_nepahmax	float8	Yes	

naytto_aanenpainetasomin	float8	Yes	
naytto_aanenpainetasomax	float8	Yes	
naytto_laja	float8	Yes	
naytto_liuos	int4	Yes	
naytto_llaskevapaajaahdytys	bool	Yes	
naytto_lvapaajaahdytysvaraus	bool	Yes	
tuoteperhe	int4	Yes	
painotyhjana	float8	Yes	
ilmanpainehavio	float8	Yes	
nesteenlammonsiirtokerroin	float8	Yes	
nesteenreynolds	float8	Yes	
ilmannopeus	float8	Yes	
aanenpainetaso125	float8	Yes	
aanenpainetaso250	float8	Yes	
aanenpainetaso500	float8	Yes	
aanenpainetaso1000	float8	Yes	
aanenpainetaso2000	float8	Yes	
aanenpainetaso4000	float8	Yes	
aanenpainetaso8000	float8	Yes	
aanentehotaso125	float8	Yes	
aanentehotaso250	float8	Yes	
aanentehotaso500	float8	Yes	
aanentehotaso1000	float8	Yes	
aanentehotaso2000	float8	Yes	
aanentehotaso4000	float8	Yes	
aanentehotaso8000	float8	Yes	
loglampotilaero	float8	Yes	
pintaalojensuhde	float8	Yes	
kokonaislammonsiirtokerroin	float8	Yes	
lahtevanilmansuhteellinenkosteus	float8	Yes	
laskentaluuppienlukumaara	int4	Yes	
kilowatinhinta	float8	Yes	
etaisyyspaadysta	float8	Yes	

etaisyyssivusta	float8	Yes	
dllnestejaahdytinok	bool	Yes	
dllnestejaahdytinvirhesanoma	var-char(255)	Yes	NULL::charactervarying
dllnestejaahdytinvirhekoodi	int4	Yes	
dllilmanlammonsiirtook	bool	Yes	
dllilmanlammonsiirtovirhesano-ma	var-char(255)	Yes	NULL::charactervarying
dllilmanlammonsiirtovirhekoodi	float8	Yes	
dllilmanvastusok	bool	Yes	
dllilmanvastusvirhesanoma	var-char(255)	Yes	NULL::charactervarying
dllilmanvastusvirhekoodi	int4	Yes	
dllaanetok	bool	Yes	
dllaanetvirhesanoma	var-char(255)	Yes	NULL::charactervarying
dllaanetvirhekoodi	int4	Yes	
dllvakiotuoterakenneok	bool	Yes	
dllvakiotuoterakennevirhesanoma	var-char(255)	Yes	NULL::charactervarying
dllvakiotuoterakennevirhekoodi	int4	Yes	
dlltoimintapisteok	bool	Yes	
dlltoimintapistevirhesanoma	var-char(255)	Yes	NULL::charactervarying
dlltoimintapistevirhekoodi	int4	Yes	
dllpuhallinok	bool	Yes	
dllpuhallinvirhesanoma	var-char(255)	Yes	NULL::charactervarying
dllpuhallinvirhekoodi	float8	Yes	
dllpatterirakenneok	bool	Yes	
dllpatterirakennevirhesanoma	var-char(255)	Yes	NULL::charactervarying
dllpatterirakennevirhekoodi	int4	Yes	
dllnestevastusok	bool	Yes	

dllnestevastusvirhesanoma	var-char(255)	Yes	NULL::charactervarying
dllnestevastusvirhekoodi	int4	Yes	
dllnesteenlammonsiirtook	bool	Yes	
dllnesteenlammonsiirtovirhesa-noma	var-char(255)	Yes	NULL::charactervarying
dllnesteenlammonsiirtovirhekoo-di	int4	Yes	
dllkosteusok	bool	Yes	
dllkosteusvirhesanoma	var-char(255)	Yes	NULL::charactervarying
dllkosteusvirhekoodi	int4	Yes	
dlllammonsiirtook	bool	Yes	
dlllammonsiirtovirhesanoma	var-char(255)	Yes	NULL::charactervarying
dlllammonsiirtovirhekoodi	int4	Yes	